Handboek

F O R T H

voor gebruik binnen de Nederlandse
gemeenschap van FORTH gebruikers

geschreven door Paul Bartholdi*)

vertaald uit het Frans door

Hans Nieuwenhuijzen en
Theo Gunsing
december 1977
Sterrekundig Instituut
Utrecht

*) Observatorium van Genève
Sauverny, Zwitserland

Handboek

F O R T H

voor gebruik binnen de Nederlandse
gemeenschap van FORTH gebruikers

geschreven door Paul Bartholdi*)

vertaald uit het Frans door
Hans Nieuwenhuijzen en
Theo Gunsing
december 1977
Sterrekundig Instituut
Utrecht

*) Observatorium van Genève
Sauverny, Zwitserland

INHOUD

-i- Voorwoord van auteur

-ii- Voorwoord van vertalers

Ø. Inleiding

1. De stacks en hoe te gebruiken
2. Arithmetische operatoren en functies
3. De woordenlijst, procedures en vocabulaires
4. Lussen en voorwaardelijke sprongen / keuze
5. De assembler, codeprocedures en de macro-assembler
6. Types van variabelen
7. Normale communicatie met FORTH
8. Virtueel en massageheugen
9. De "editor"

A. Basis vocabulaires

B. Gebruikers vocabulaires

C. Locale implementatie

D. Koppeling van routines die in een andere taal zijn geschreven

E. Microprogrammering

F. Communicatie met CAMAC

G. Systeembeveiliging

H. Beveiliging data bestanden

I. Monitor voor meerdere taken

J. Real-time monitor

K. Monitor voor meerdere gebruikers

T. Hulpmiddelen voor documentatie en programma-ontwikkeling

U. Elementair gebruik van FORTH aan de console

V. Uitgewerkte voorbeelden

W. Metaforth (generator van FORTH, in FORTH)

X. Forth programmatheek

Y. Litteratuur

Z. Index

VOORWOORD van auteur

Aan het einde van de jaren zestig heeft Charles Moore, van het N.R.A.O. te Kitt Peak FORTH ontworpen. De algemene toepassingen van FORTH in Amerikaanse, Europese en Australische observatoria zijn echter zeer recent. Intussen hebben Charles Moore en Elisabeth Rather een onderneming*) opgericht met het oogmerk turn-key FORTH systemen te leveren aan observatoria, ziekenhuizen, industrie, etc.

Hoewel FORTH tot nu toe voor minicomputers was aangepast, zijn nu ook toepassingen gerealiseerd op grote machines als b.v. de IBM 360/50, CDC 6400 en UNIVAC 108 en op microprocessoren. RCA heeft zelfs FORTH geselecteerd als taal voor haar COSMAC microprocessor.

In augustus 1976, op een IAU-bijeenkomst in Grenoble, ontdekten we dat reeds een dozijn Europese astronomen met FORTH werkte. Wij besloten een FORTH gebruikersclub op te richten voor *héél* Europa en de Verenigde Staten met het doel om de verschillende gebruikersversies op elkaar af te stemmen, een minimumstandaard te definiëren en de overdraagbaarheid van systemen te bevorderen.

De groep komt twee à driemaal per jaar bijeen en publiceert de bijdragen van haar leden.

De secretaris van de Europese FORTH gebruikers club is:

Dr. Hans Nieuwenhuijzen
Sterrekundig Instituut
Zonnenburg 2
Utrecht.

In het eerste jaar van het bestaan van de groep is o.a. een zeker niveau van standaardwoorden gedefinieerd en geaccepteerd, en is een systeembeveiliging gerealiseerd die in belangrijke mate de kans op fatale fouten verkleint.

---

*) FORTH Inc., 815 Manhattan Ave., Manhattan Beach, CA 90266, U.S.A.

Forth bezit een kwaliteit die noch in de astronomie noch in de informatica duidelijk wordt gezien omdat diegenen, die het leren kennen, fanatiek vóór of tegen Forth worden. Dit gedrag verhindert de werkelijke hoedanigheid van FORTH te zien, welke als volgt is: een zeer machtig gereedschap in bepaalde domeinen, beperkt in andere, in ieder geval een tussenschakel (interface) tussen systeem en gebruiker (intelligent verondersteld) welke een zeer fundamentele rol speelt.

Forth is niet uniek; omstreeks dezelfde tijd zijn veel andere talen ontstaan met ongeveer dezelfde doelstellingen.

De zorg van de informatica over deze talen is gericht op de hiërarchieke gestructureerde programmering. FORTH bevindt zich midden in deze maalstroom van verschillende standpunten, maar ondanks dat heeft het een aantal zeer markante eigenschappen, gericht op data acquisitie systemen en controle van instrumenten.

In dit handboek zal ik de verschillende aspecten van FORTH behandelen, de sterke maar ook de zwakke, welke men kan ondervinden bij het gebruik.

Genève, juni 1977. Paul Bartholdi

VOORWOORD van vertalers, bewerkers

Door het werk van Paul Bartholdi te vertalen hopen wij een duidelijk beeld te scheppen van FORTH voor de Nederlandstalige gebruikers.

Wij zullen ingeburgerde Engelse uitdrukkingen normaal gebruiken, zodat er een aansluiting blijft bestaan bij de overheersende Engelse litteratuur.

Hans Nieuwenhuijzen
Theo Gunsing

Ø.1 INLEIDING

FORTH is geen - uitontwikkelde taal
- systeem
- vertaler
- "interpreter"
- virtuele machine
- assembler
- contrôletaal
- besturingstaal,

maar van alles een deel en soms een beetje meer. Dit is een vreemde manier van presenteren (negatief), maar het is zo gedaan om direct de diverse aspecten te tonen van FORTH zonder dat er eerst mee moet worden gewerkt.

FORTH is ook geen uitgekristalliseerde constructie, die één keer en daardoor voor altijd gedefinieerd is. Zekere basiselementen zijn duidelijk wel ondubbelzinnig gedefinieerd, maar andere zijn implementaties van een uit te voeren taak. Niettemin is de overdraagbaarheid van het ene systeem naar het andere òf van de ene computer naar de andere zeer groot. Bij dit soort acties is slechts een strikt minimum aan aanpassingen nodig. Oorspronkelijk is FORTH geconcipieerd als een zelfstandig geheel, zonder ook maar een verbinding te hebben met een systeem of bestaande taal.

Om FORTH goed te kunnen bedrijven moet aan een aantal minimum voorwaarden voor de hardware worden voldaan, t.w.:

- een centrale verwerkingseenheid (processor)
- een massa geheugen (magneetband, schijf, "floppy disc")
- minder geschikt papierband
- een console (besturingstelex) (of meerdere)
- [- een of meerdere instrumenten om te sturen of uit te lezen door het systeem] .

Een belangrijk oogmerk bij de constructie van FORTH is geweest het mogelijk te maken om een niet in de informatica ingewerkte snel in staat te stellen om toch met een computer zijn meetinstrumenten te beheersen.

Dankzij het werk van Svend Lorensen (ESO) is FORTH uitgebreid met een mogelijkheid om zonder een zelfstandig FORTH massageheugen ingepast te worden in een bestaand "operating system" zoals RTE II/III van Hewlett Packard. Er zijn bezwaren aan te voeren tegen zo'n toepassing van FORTH, maar het maakt wel het inpassingsvermogen van FORTH duidelijk.

Ø.2 FORTH als "interpreter"

FORTH moet op zo'n manier interactief gebruik mogelijk maken, dat de gebruiker direct het resultaat van zijn opdrachten kan zien inclusief foutmeldingen van dingen die niet gedaan mogen worden.

Een "interpreter" doet dit werk. De opdrachten worden direct uitgevoerd zonder een aparte compilatieslag. BASIC is hier een voorbeeld van met de volgende beperkingen:

- beperkt aantal instructies
- traag in uitvoering omdat elke "statement" eerst berekend en daarna vertaald moet worden voordat de uitvoering plaatsvindt.

Bij FORTH is dit op twee manieren opgelost:

- een zeer uitgebreide en elastische manier van invoeren van instructies
- toestaan van compilatie van procedures, gebruik makend van dezelfde instructies als de "interpreter" zelf doet.

De overgang van de ene modus naar de andere gaat uitzonderlijk snel en simpel. Deze eigenschappen kunnen worden gecombineerd tot directe uitbreidingen van de interpreter en zijn dan verder ook gewoon bruikbaar.

Anders gezegd:

Een instructie, ingegeven aan de console, wordt geïnterpreteerd en direct geplaatst achter de laatst toegevoegde procedure, instructie, in de bibliotheek. Was dit een getal, dan wordt dit op de operationele "stack" geplaatst. Betreft het een opdracht, dan wordt uit de bibliotheek het adress opgezocht van de routine met deze naam. Met dien verstande, dat van achter naar voren wordt gezocht, zodat altijd de jongste definitie wordt gebruikt. Alleen op dit moment wordt in de bibliotheek gezocht, een moeizaam werk voor de interpreter.

Als men snel wil werken dan moet men de interpreter mode dus niet al te veel instructies geven.

Anderzijds is de interpreter een hulpmiddel dat bijzonder nuttig is gedurende definitie- en testfase.

1.Ø De verschillende stacks van FORTH

FORTH gebruikt drie verschillende stacks bij de uitvoering van zijn taak. Deze zijn:

1. De arithmetische of operationele stack;
2. De "return stack";
3. De bibliotheek.

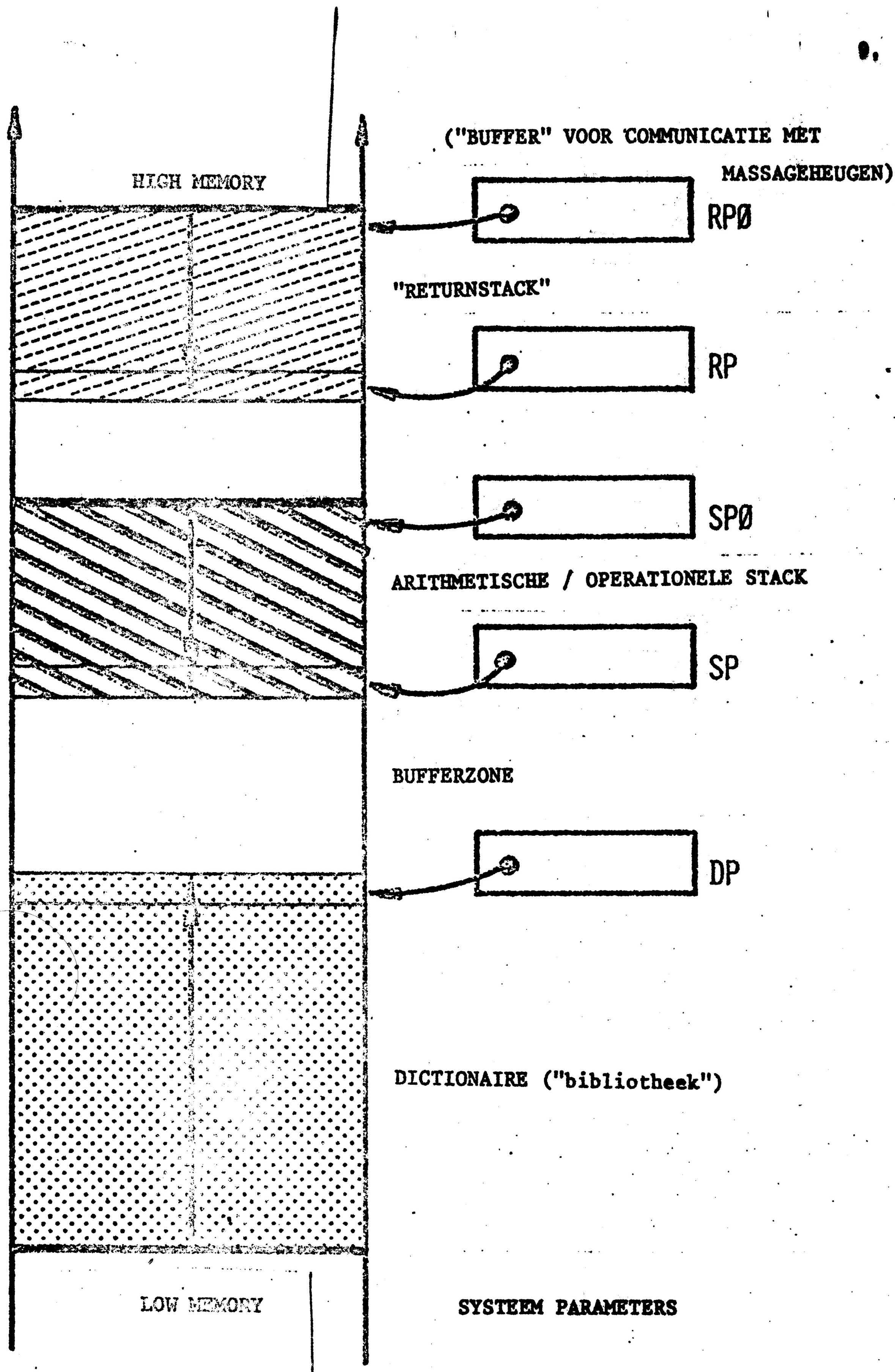
De operationele stack bevat alle variabelen en parameters die niet expliciet zijn gedefinieerd. Algemeen geldt:

Alle operatoren vinden hun operand(en) aan de top van deze stack, daar worden zij weggehaald en indien van toepassing vervangen door hun resultaten.

De **return** stack bevat in principe het adres van de procedure waarmede verder zal worden gegaan na beëindiging van de huidige werkzame procedure (instructie volgend na aanroep van genoemde procedure).
In het merendeel van de implementaties bevat deze stack ook de parameters van de lussen (DO-loops). De bibliotheek, als laatste, bevat de verzameling van instructies, die in de vorm van een of meerdere lijsten zijn vertaald.

Opmerkingen :

- de lengte van de returnstack is vast ($3Ø_8$ meestal).
- de positie en afmeting van de bufferzône is dynamisch en varieert tijdens de uitvoering van taken. Meestal is er geen bescherming tegen het overschrijven van de ene stack door de andere.
- de bufferzône wordt gebruikt gedurende de compilatie en interpretatie en moet minimaal een dertigtal woorden bevatten.



PLAATSING VAN DE DRIE STACKS IN HET GEHEUGEN

Opmerking: De plaatsing van de stacks hangt af van de installatie. (Er zijn machines met de operationele stack in low memory.)

1.1 De arithmetische/operationele stack

Deze stack zal in het vervolg kortweg "de stack" worden genoemd. Er zijn een paar operatoren nodig om de top van deze stack te behandelen. Het doel van deze operatoren is de parameters in de goede volgorde te plaatsen voor de komende operatoren. Dit is waarschijnlijk het stuk van FORTH, waarmee men de meeste moeilijkheden ondervindt en waarin de meeste fouten worden gemaakt.

Opmerking 1 :

- de operatoren moeten hun parameters (operanden) aan de top van de stack vinden;
- deze operanden worden daar vernietigd en - indien van toepassing - vervangen door het resultaat verkregen met de operatoren.

a) "Stack pointers"

SP	bevat	het adress van de top van de stack
SP@	is	het adress van de top van de stack
SPØ	is	het adress van de bodem van de stack.

Opmerking 2 :

- De stack groeit van high naar low memory, terwijl de return stack van low naar high memory gaat. (Er zijn echter uitzonderingen.)

b) Verwijderen van één of meer parameters van de stack

DROP	verwijdert een parameter van de stack
n NDROP	verwijdert n parameters van de stack.

c) Verdubbeling van parameters op de stack

DUP	verdubbelt de parameter op de stack
.	(ABD--- op de stack wordt AABC---)
2DUP	idem voor parameter met twee locaties.

d) Copiëren vanuit de stack

OVER	copieert een parameter naar de top van de stack
2OVER	idem voor een dubbele parameter (2-woords parameters)
n NOVER	idem voor een n-voudige parameter.
n ~~8~~ PICK	copieert de n[de] waarde op de stack boven op de stack.

e) Permutaties aan de top van de stack

SWAP	ruilt de bovenste twee parameters op de stack van plaats
2SWAP	idem voor dubbele parameters.
ROT	permuteert de drie variabelen aan de top van de stack
2ROT	idem voor dubbele parameters.

f) Communicatie tussen stack en geheugen

a	vervangt het adress op de stack door de inhoud van dat adress
D*a*	idem voor een dubbelwoord parameter (een adress op de stack!)
!	stopt het volgende woord van de stack in het adress boven op de stack
D!	stopt de volgende twee woorden van de stack op de adressen, aangegeven boven op de stack (het gegeven adress en de daarop volgende) (één adres op de stack).
+!	telt het tweede woord van de stack op bij het adress aangegeven op de top van de stack.
-!	idem, maar i.p.v. "telt op" nu "trekt af" .
1+!	telt 1 bij de inhoud van het adress op de stack
1-!	trekt 1 af van de inhoud van het adress op de stack

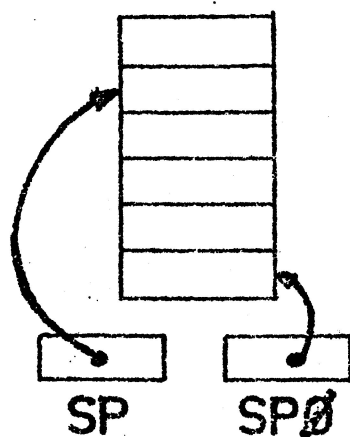
1+	telt 1 op bij de top van de stack
1-	trekt 1 af van de top van de stack
ØSET	maakt de inhoud van het woord met het adress op de stack gelijk aan Ø
1SET	idem, met 1 i.p.v. Ø .

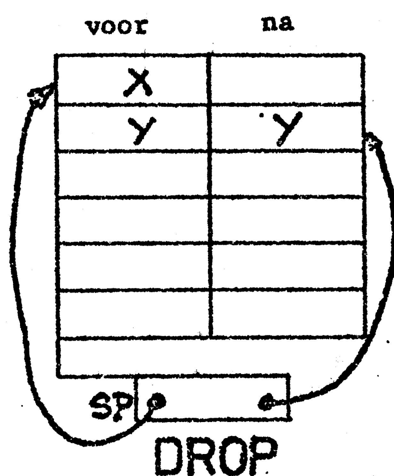
Opmerking :

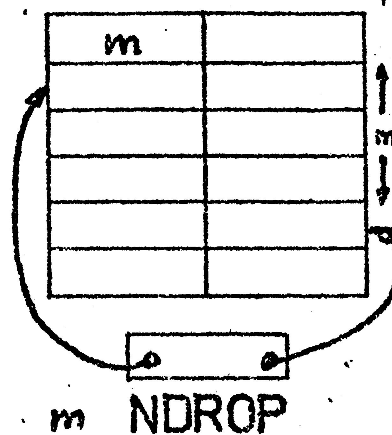
- Daar SP in een indexregister kan zijn geplaatst is deze alleen bereikbaar gemaakt met: SP@ , SP! en SP+!

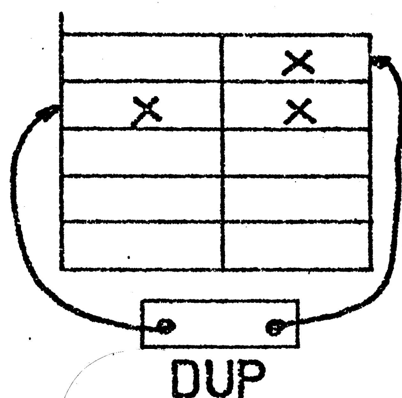
Zie voor level Ø woorden blok 390 ev
blok 15, 19 ev

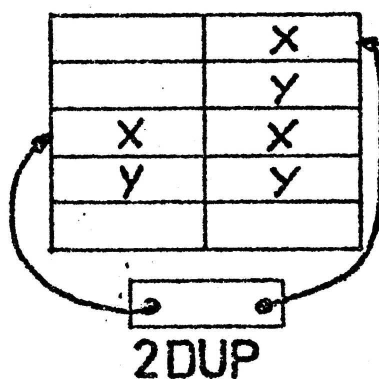
De 32 bit operatoren vanaf 1900

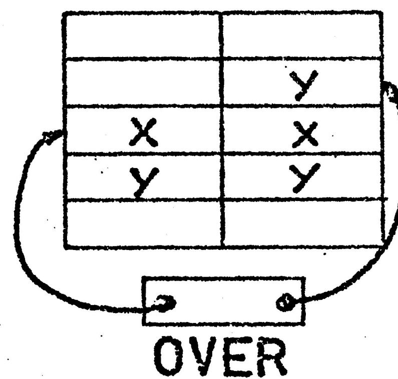

SP
SPØ

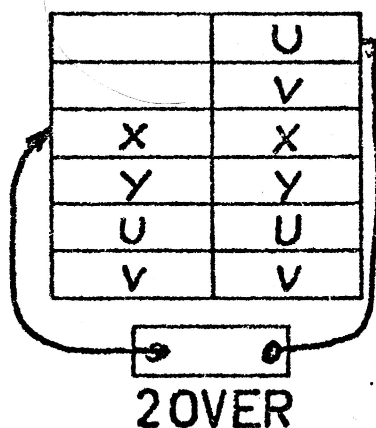

voor
na
X
Y
Y
SP
DROP

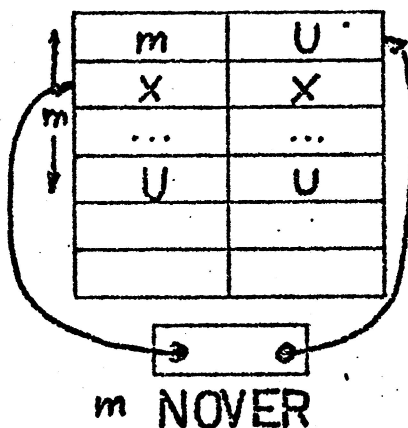

m
m
m NDROP

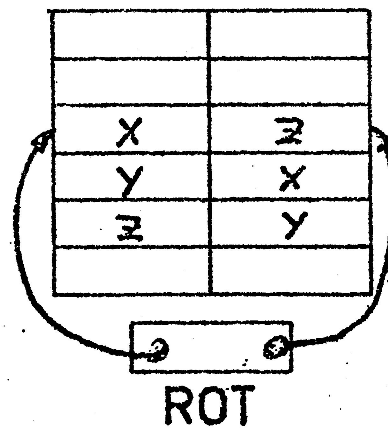

X
X
X
DUP

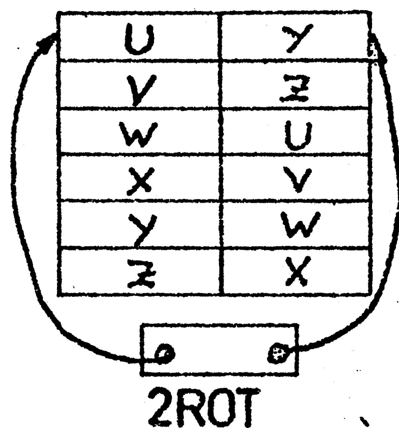

X
Y
X
X
Y
Y
2DUP

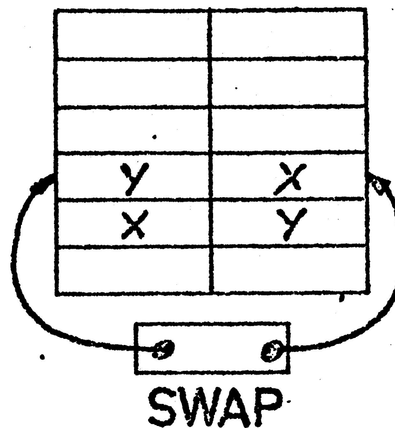

Y
X
X
Y
Y
OVER

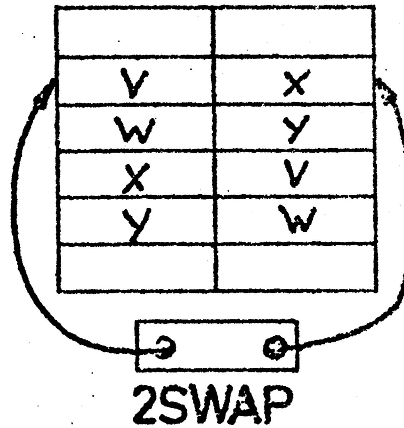

U
V
X
X
Y
Y
U
U
V
V
2OVER


m
U
X
X
m
...
...
U
U
m NOVER

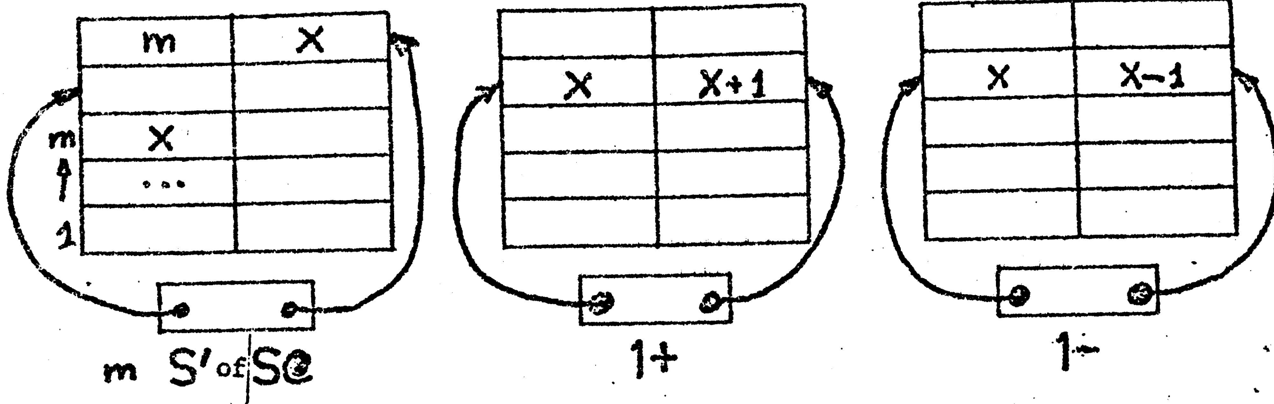

X
Z
Y
X
Z
Y
ROT

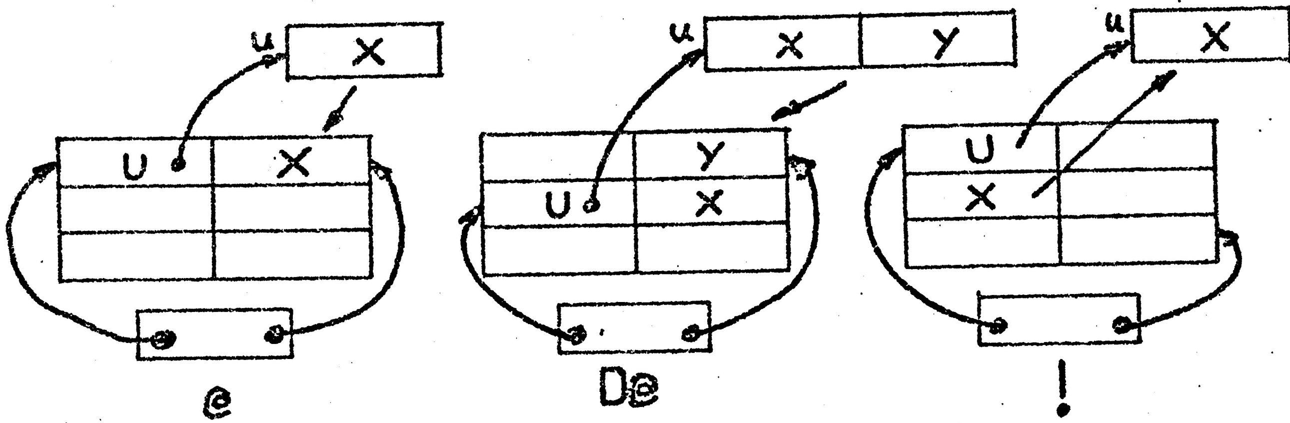

U
Y
V
Z
W
U
X
V
Y
W
Z
X
2ROT

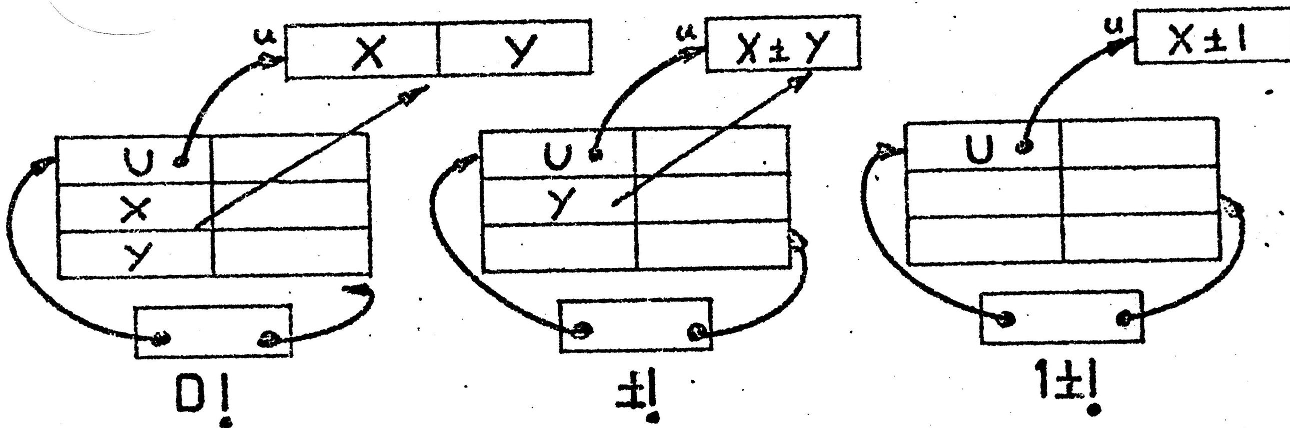

Y
X
X
Y
SWAP

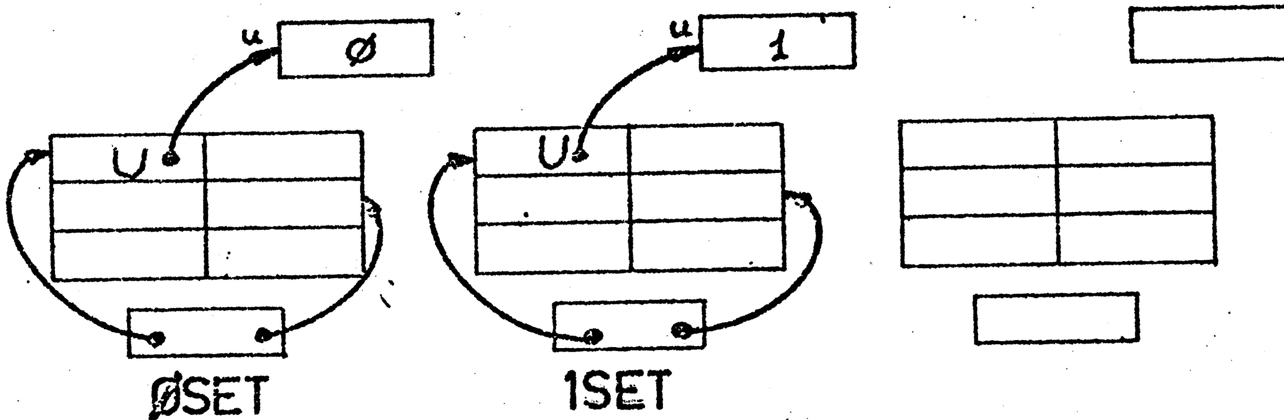

V
X
W
Y
X
V
Y
W
2SWAP

Van boven af





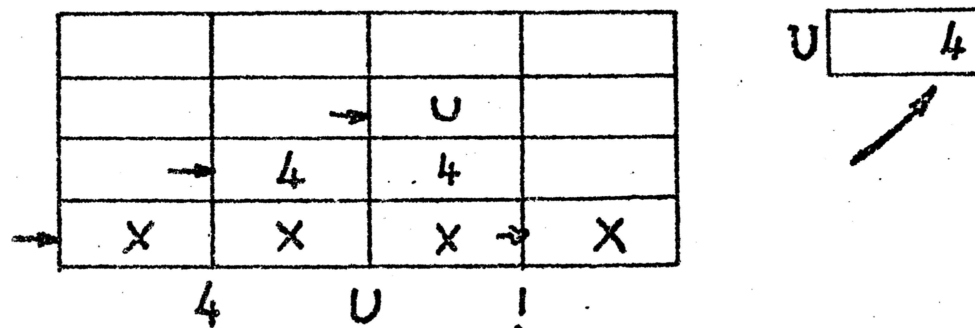




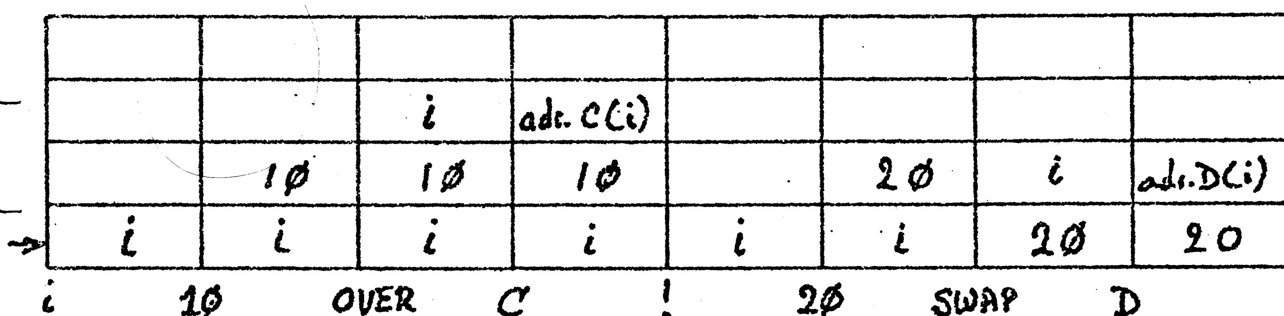
Voorbeelden :

Het is vaak moeilijk om te volgen wat er op een stack gebeurt. Tekeningen zijn een erg goed hulpmiddel hierbij. In de hierna volgende tekeningen is de top van de stack boven getekend, de instructies zijn er boven gegeven en bovendien onderaan van links naar rechts.

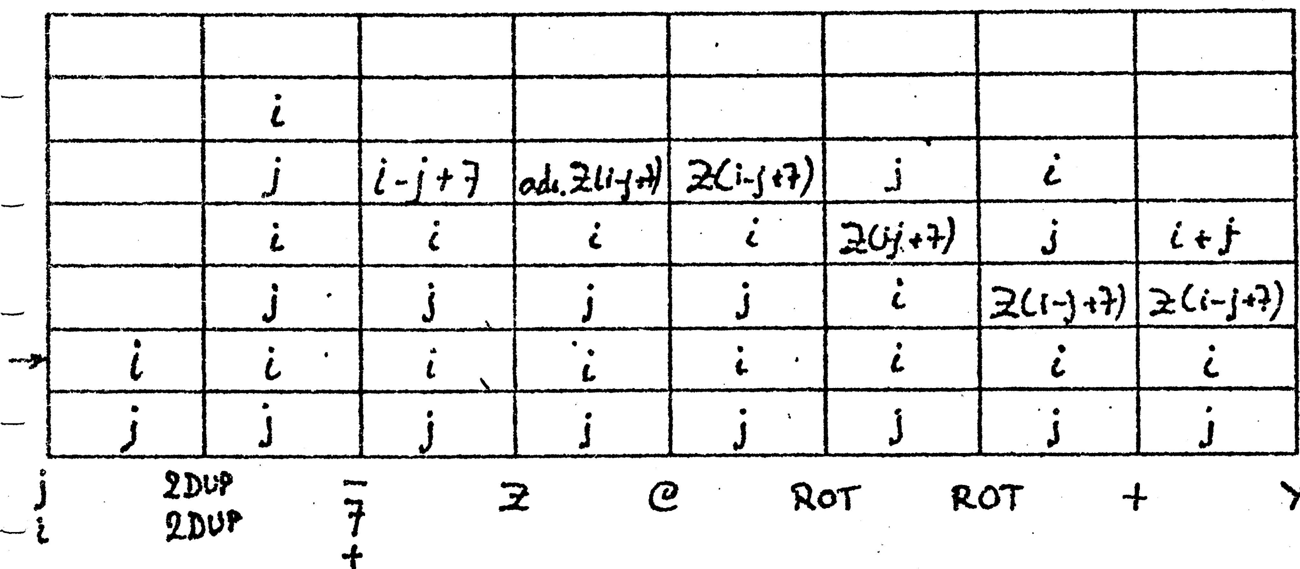
a. Men wil de variabele (INTEGER) U de waarde 4 geven.



b. Men heeft al een waarde i op de stack en wil 10 plaatsen in C(i) en 20 in D(i) : (C en D zijn ARRAYS).



c. i en j zijn op de stack en men wil berekenen X(i) AND Y(i+j) AND Z(i-j+7)



adr. Y(i+j)	Y(i+j)						
Z(i-j+7)	Z(i-j+7)	Y∧Z	i	adr. X(i)	X(i)		
i	i	i	Y∧Z	Y∧Z	Y∧Z	X∧Y∧Z	j
j	j	j	j	j	j	j	X∧Y∧Z

Y   @   AND   SWAP   X   @   AND   SWAP   DROP

d) i en j zijn op de stack en men wil X(i), Y(i+j) en Z(i+7-j) gelijk aan Ø maken.

	i				(j)			
	j	i+j	adr. Y(i+j)		i-j	Z(i-j+7)		
i	i	i	i	i	i	i	X(i)	
j	j	j	j	j	j	j	j	j

OVER OVER ou 2DUP	+	Y	ØSET	2DUP −	7 + Z	ØSET X	ØSET

1.2 De return stack

De return stack bevat aan de top het adres waarvan de bijbehorende procedure wordt uitgevoerd na het beëindigen van de actuele (maakt recursiviteit mogelijk). Op het merendeel der installaties mag deze stack ook worden gebruikt als tijdelijke opslag geassocieerd met de procedure. In het bijzonder is dit het geval voor de DO loop waarvoor de parameters (tellers) in het algemeen op deze stack worden geplaatst.

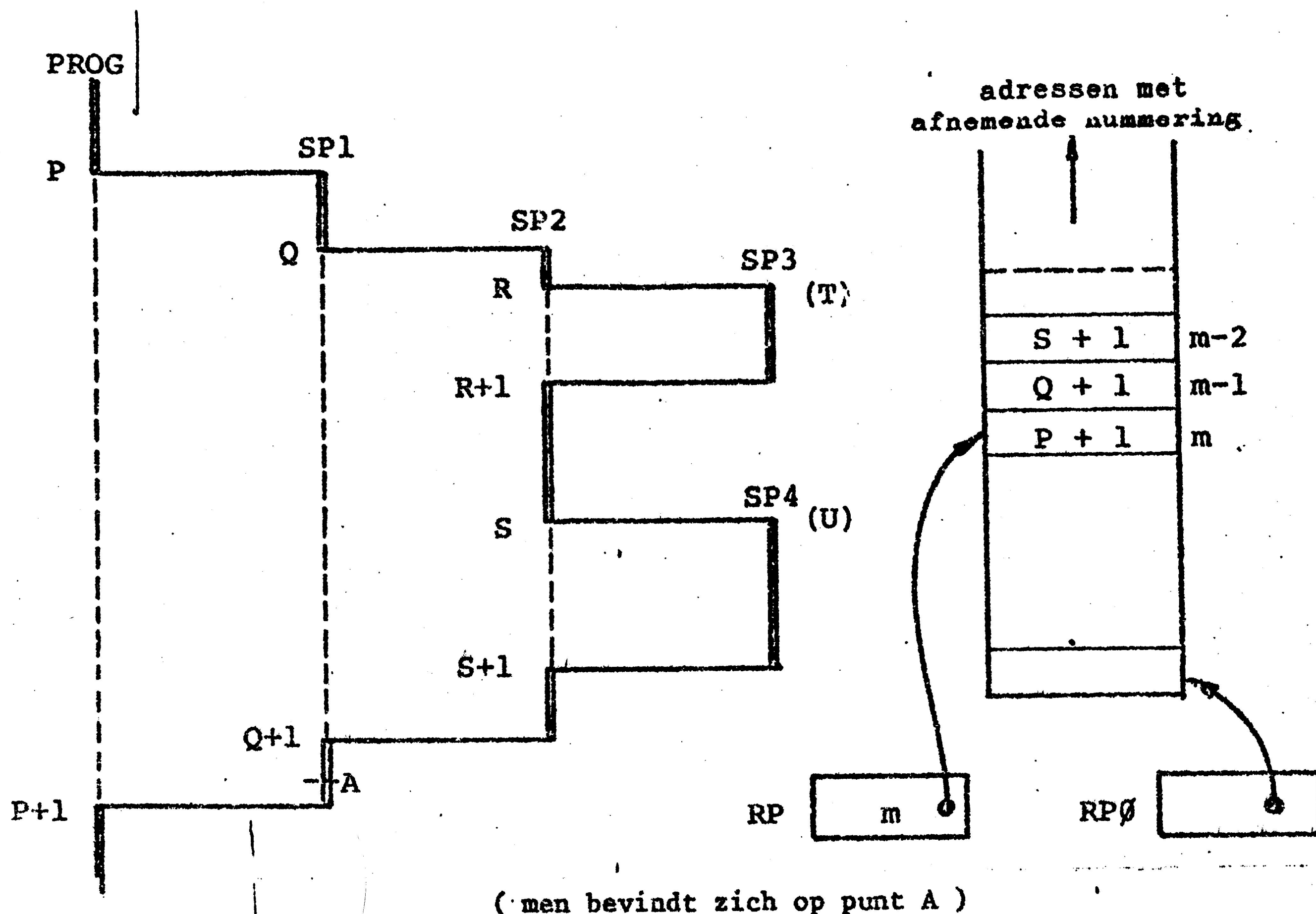
Het gebruik als tijdelijke opslagplaats moet met de grootste voorzichtigheid worden gehanteerd aangezien het systeem niet kan onderscheiden of er een parameter of return address op de stack staat.

Anderzijds is de capaciteit veel kleiner dan die van de operationele stack en overschrijvingen komen vaak voor. Deze twee typen fouten zijn voor 90% de oorzaak van een "hang up" van FORTH. Het beveiligingssysteem, geïntroduceerd in 1977, tracht deze fouten al in de compilatiefase te voorkomen.

Pointers voor de return stack

RP	bevat het adres van de pointer naar de top van de return stack
RPØ	bevat het adres van de pointer naar de plaats van de basis van de return stack
de operator RP@	zet de inhoud van RP op de stack
de operator RP+!	telt de inhoud van de top van de stack bij RP op.

N.B. Als RP aanwezig is in een register kan hij niet anders dan met bovenstaande twee operatoren worden bereikt.



(men bevindt zich op punt A)

PROG roept SP1 in P en zet P+1 op de return stack,
SP1 roept SP2 in Q en zet Q+1 op de return stack,
SP2 roept SP3 in R en zet R+1 op de return stack.

Terugkeer vanuit SP3 geschiedt via de top van de return stack, zijnde R+1 . Verderop in S wordt SP4 door SP2 aangeroepen en SP4 zet S+1 op de return stack. Terugkeer uit SP2 geeft Q+1 aan de top, dan P+1 na terugkeer in SP1 etc.

<u>Opmerkingen</u> : 1. Deze manier van "return address" manipulatie maakt het mogelijk om procedures recursief aan te roepen zonder complicaties.

2. De return stack groeit evenals de operationele stack naar "low memory". (Er zijn echter enkele uitzonderingen.)

<u>N.B.</u> De besturing van "return addressen" op een stack is nodig maar niet voldoende voor recursiviteit. Om dit te bereiken moeten ook alle variabelen en parameters van de locale procedure op dezelfde manier worden behandeld. Zie hoofdstuk 3.4.

2.1 Rekenkundige bewerkingen

FORTH kent verschillende getaltypes, o.a.

16 bits integers (enkel woord)
32 bits integers (dubbel woord)
integerfracties (tussen -2.Ø en +2.Ø) (enkel woord)
32 à 48 bits floating point getallen.

Het aantal bits hangt typisch af van de soort processor.

Elk type variabele wordt bewerkt door een specifieke rij operatoren gekenmerkt door een letter vlak voor de operator.
Deze zijn voor het optellen (+)

integer	+
dubbele integer	D+
fractie	,+
floating point	F+

De programmeur moet zelf zorgen voor de koppeling van het type variabele met de juiste operator. Dit wordt niet door het systeem gecontroleerd.

Opmerking 1 :

- Alle operatoren vinden hun operanden (*niet* de adressen) op de stack alwaar zij deze vervangen door de resultaten van de operatie. Dit impliceert voor FORTH de reversed polish notation zoals bij zekere calculators (HP, NOVUS, SINCLAIR, etc.).

  Het is hierbij practisch nooit nodig (tijdelijk) tussenresultaten op te slaan in een buffergeheugen.

  Een strategie om aan deze methode te wennen is de volgende:

  1. Begin met altijd zonder tussenhaakjes te werken;
  2. Laat tussenresultaten op de stack. Het zal dan blijken dat, altijd op het moment dat zij nodig zijn, zij op de natuurlijke plaats worden gevonden.

Opmerking 2 :

- De Poolse notatie is niet erg natuurlijk en is moeilijk terug te lezen zonder een terugvertaling naar de conventionele algebraïsche schrijfwijze. Anderzijds heeft ze een aantal voordelen boven de conventionele schrijfwijze:

  - er zijn geen haakjes meer nodig;
  - de afwezigheid van twijfel in de volgorde van de bewerkingen;
  - de mogelijkheid om complexe operatoren te maken die op bijv. drie operanden werken. (Zie hiervoor bijv. de hierna volgende bewerkingen */ en /MOD).

2.2 Lijst van arithmetische operatoren

	standaard Integer	Fractie	Dubbel integer	Floating point	Nederlandse omschrijving
	a	a	Da	Fa	Vervangt het adres op de stack door de inhoud van dat adres.
	!	!	D!	F!	Op de top van de stack staat een adres en daaronder een getal. Deze operator stopt het getal in het adres.
	+	+	D+	F+	Twee getallen op de stack worden vervangen door hun som.
blok 15	-	-	D-	F-	Twee getallen op de stack worden vervangen door hun verschil.
blok 29	*	,*	D*	F*	Twee getallen op de stack worden vervangen door hun product.
	/	,/	D/	F/	Twee getallen op de stack worden vervangen door hun quotiënt.
	MOD		DMOD	FMOD	Als / . Nu echter rest op de stack.
	/MOD				Als MOD . Nu quotiënt boven op de stack, de rest daaronder.
	ABS	ABS	DABS	FABS	Verandert een getal op de stack tot zijn absolute waarde.
	MINUS	MINUS	DMINUS	FMINUS	Keert het teken om van het getal op de stack.
	MAX	MAX	DMAX	FMAX	Laat het maximum van twee getallen op de top van de stack.
	MIN	MIN	DMIN	FMIN	Laat het minimum van twee getallen op de top van de stack.
	*/				Vermenigvuldigt twee getallen op de stack (net onder de top) en deelt het resultaat door het getal aan de top van de stack met behoud van de dubbele precisie gedurende deze bewerking.
	1+				Telt één op bij het getal op de stack.

1-			Trekt één af van het getal op de stack.
1+!			Telt één op bij de inhoud van het adres dat gegeven is op de top van de stack.
1-!			Als 1+! , maar nu met aftrekken.
FLOAT	DFLOAT		Zet een integer om naar zijn equivalent in floating point.
		FIX of DFIX	Zet een floating point getal om naar een enkele of dubbele integer.

<u>Opmerking</u> :

- Al deze operatoren vernietigen de parameters op de stack en plaatsen hun resultaat op de stack indien aangegeven.

  De volgorde van parameters is dezelfde als wanneer de operator er normaal was tussen geplaatst.

Voorbeeld : (A B -) = (A-B) ; (A B C */) = (A*B/C) .

2.3 Lijst van logische operatoren

Integer	Fractie	Dubbel integer	Floating point	Nederlandse omschrijving
=	=	D=	F=	Vervangt de twee parameters op de stack door het resultaat van de logische bewerking bij het uitvoeren van de vergelijking:
#	#			
<	<			
>	>			
Ø<	Ø<			Als het waar is dan ≠Ø
Ø=	Ø=			Als het onwaar is dan =Ø .
AND				Vervangt de parameters door hun logisch product.
OR				Vervangt de parameters door hun logische som.
XOR				Vervangt de parameters door hun logische exclusive or .
BNOT of LNOT				Vervangt de parameter door zijn logische inverse .

Opmerking :

- BNOT en LNOT zijn gelijk aan Ø= en zij moeten alleen worden gebruikt als er kans is op vergissingen met een echte arithmetische vergelijking.

2.4 Arithmetische functies (van floating point getallen (reals))

In hetgeen hierna komt zijn de volgende conventies gebruikt:

a	geeft een hoek in radialen
i, j, k	zijn integers
f	een real
f+	een positieve real (positief definiet)

Gebruik	Resultaat	Omschrijving
i j ↑	k	i tot de macht j
f i F↑	f	f tot de macht i
f+ ALOG	f	natuurlijke logarithme van f+
f+ ALOG1Ø	f	briggse logarithme van f+
f ATAN	a	arctangens van f
a COS	f	cosinus van a
f EXP	f+	e^f
a SIN	f	sinus van a
f+ SQRT	f+	wortel van f
PI	f	π

3.Ø De woordenlijst

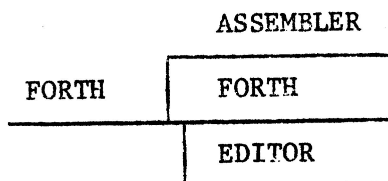
De woordenlijst (of dictionaire) bevat de verzameling procedures en gegevens welke op een zeker moment aanspreekbaar zijn. Alhoewel ze een naam hebben zijn ze niet alfabetisch gerangschikt, maar in de volgorde waarin zij zijn gedefinieerd. In deze woordenlijst is een aantal vocabulaires, die bestaan uit een verzameling woorden, welke bereikbaar zijn via pointers. De pointer NIL (=Ø) geeft het einde aan van zo'n vocabulaire of van de woordenlijst. (Zie 3.8)

De vocabulaires kunnen aan elkaar worden geregen door het laatste woord van de ene te laten wijzen naar het eerste van de andere.

Tijdens de interpretatie- of compilatiefasen kan slechts één vocabulaire tegelijkertijd worden aangesproken, maar wel mag de gehele woordenlijst worden gebruikt (vocabulaires door elkaar) tijdens de executiefase.

Tenslotte kan een vocabulaire worden gedefinieerd in de FORTH vocabulaire, zodat ze altijd beschikbaar is. Ook kan ze in een andere vocabulaire worden geplaatst zodat ze dan gedurende activiteiten in de omhullende vocabulaire bereikbaar blijft.

De vocabulaires vormen een boomstructuur waarvan de FORTH-vocabulaire de stam is
Normaal bevat FORTH minstens drie vocabulaires:

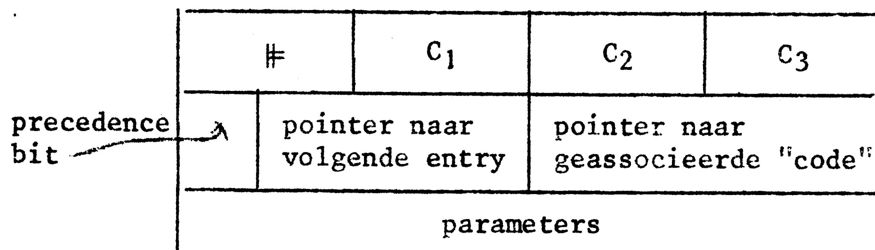


3.1 Structuur van een element, woord uit de woordenlijst

Elk element van de woordenlijst is opgebouwd uit twee gebieden:

- een header (of kop)
- een parameterveld (parameters, instructies, etc.)

De header heeft altijd dezelfde vorm en wel zoals in de onderstaande figuur is gegeven:



De header neemt vier woorden in op miniprocessoren (16 bits woorden) of minder als de woordlengte van de machine het toestaat.

Elke entry heeft een naam die voornamelijk naar de lengte wordt geïnterpreteerd, en wel:

is het aantal karakters waaruit de naam bestaat

C_1 C_2 C_3 de eerste drie karakters van het woord.
(Er zijn systemen waarbij meer karakters worden gebruikt, tot nu toe 256 als maximum).

Het precedence-bit geeft aan of de entry uitgevoerd of gecompileerd zal worden in de nieuwe definitie (zie 3.3) (als =1, dan uitvoeren.)

De eerste pointer wijst naar de voorgaande entry in de woordenlijst. Tijdens interpretatie of compilatie wordt met behulp van deze pointer de woordenlijst afgezocht totdat een entry wordt gevonden met # C_1 C_2 C_3 welke overeenstemt met het gezochte. (Het is mogelijk om een stuk van de woordenlijst of een vocabulaire onbereikbaar te maken (te beveiligen) door de pointer te veranderen, waardoor het effect wordt verkregen alsof er in deze lijst is geknipt.

De pointer naar de geassocieerde code mag worden opgevat als een "machine instructie" welke met de entry is verbonden.

Op het moment van interpretatie of uitvoering is dit de geassocieerde code welke uitgevoerd gaat worden en bij compilatie wordt het adres waar deze pointer staat in de woordenlijst geplaatst (net als een machine instructie).

De header heeft een vaste lengte terwijl het parameterveld een wisselende lengte heeft of zelfs geheel kan ontbreken.

Men vindt in het parameterveld bijvoorbeeld de waarde van een constante, een variabele, een vector of matrix of een lijst met adressen van procedures of machine-instructies. In het laatste geval wijst de pointer van de geassocieerde code naar het woord dat volgt op de pointer. (Zie 3.2)

Opmerkingen over de naam in een header:

1. Elk ASCII karakter mag worden gebruikt op elke plaats.

2. Twee namen met dezelfde lengte en gelijke karakters op de eerste drie plaatsen zijn voor FORTH identiek (synoniemen).

3. In het geval van *synoniemen* (2) kan de laatste van de voorgaande gebruik maken, maar de voorgaande is vervolgens niet meer vindbaar voor FORTH bij zoeken in de woordenlijst. (Zie 3.4)

3.2 De procedures

De bijzondere manier van schrijven en het gebruik van procedures is een fundamentele karakteristiek van FORTH. Zij maken een hiërarchische programmering heel eenvoudig mogelijk.

Een nieuwe procedure wordt op de volgende manier ingevoerd:

```
:  < naam van de procedure >   < body van de procedure >  ;
```

De naam van de procedure is vrij. Alleen moet worden bedacht dat er een kans bestaat op het vormen van synoniemen.

De "body" van de procedure is een lijst van namen van andere procedures in de volgorde waarin zij moeten worden uitgevoerd.
(Herhalings- of keuze-operatoren kunnen de volgorde van uitvoering beinvloeden (zie hoofdstuk 4)).

Voorbeeld : Een astronomisch fotometrisch experiment bevat:
- een nulpunts ijking,
- opmeting van een ster,
- opmeting achtergrond van de hemel,
- afdrukken van de resultaten.

Men kan met de procedure MEET definiëren als

```
: MEET NULPUNT STER HEMEL PRINT ;
```

Met de procedures NULPUNT etc., welke op hun beurt op dezelfde manier gedefinieerd kunnen zijn:

```
: NULPUNT    C-U RESET    C-B RESET    C-U RESET ;
: STER    CENTREER
          FILTER-U   INTEGER C-U !
          FILTER-B   INTEGER C-B !
          FILTER-V   INTEGER C-V !  ;
```

etc.

Deze manier van aanpak van procedures is bekend onder de naam van "TOP DOWN DESIGN" of "STEPWISE REFINEMENT". De basisgedachte bij deze werkwijze is het schrijven van korte procedures die in een oogopslag te overzien zijn zonder zich zorgen te maken hoe op bepaalde andere niveaux iets wordt gerealiseerd. Zo zal men in het voorbeeld op het moment dat men STER wil gebruiken zich alleen bezighouden met het feit dat men een ster wil observeren. Hoe dat wordt uitgevoerd is van later zorg. Dat is eerst van belang op het moment dat STER wordt gedefinieerd. Uiteindelijk zullen er procedures worden opgesteld die in detail beschrijven wat er moet gebeuren.

Opmerking :

- In FORTH kan men een procedure eerst aanroepen nadat ze is gecodeerd. Zo zal de procedure MEET niet de eerst maar de laatst te definiëren procedure zijn. Dit is het tegenovergestelde van de beschreven techniek en wordt ook wel "BOTTOM UP CODING" genoemd met de volgende twee grote voordelen, welke hierna zullen worden genoemd. Eerst nog even: de analyse gaat van grote lijnen naar details, terwijl de codering in omgekeerde volgorde gaat, $\left(\frac{\text{Analyse}}{\text{TOP DOWN}} - \frac{\text{Codering}}{\text{BOTTOM UP}} \right)$.

  De voordelen zijn:

  1. Zodra een procedure (in detail) is gecompileerd, kan ze direct worden getest. In ons voorbeeld plaatst de procedure FILTER-U het filter U in de gewenste positie. Het is uiterst eenvoudig om dit commando te geven en te controleren of de procedure goed wordt uitgevoerd. Ingeval van twijfel is de procedure zo simpel dat fouten makkelijk worden ontdekt. In het vervolg stellen we dat alle procuderes op een lager niveau getest zijn en correct bevonden.

  2. Het is zonder meer mogelijk om de procedures, onafhankelijk van het niveau, door verschillende personen te laten schrijven.
     In ons voorbeeld kunnen de elementaire procedures, die de fotometer besturen, worden geschreven door de technicus die de computeraansluiting heeft gerealiseerd. De procedure MEET en de functies van het waarnemingsprogramma kunnen in dit voorbeeld door de astronoom zelf worden geschreven zonder dat hij zich hoeft in te werken in de hardware.

Een ander aspect van "TOP DOWN" programmering is de theoretische manier van aanpak die zich er in eerste instantie niet druk om maakt of de procedures wel kunnen worden gerealiseerd. (Het probleem wordt vooruitgeschoven.) Ondanks alle voorzorgen kan een contradictie ontstaan tussen het algehele idee en de implementatie in detail. De combinatie "TOP DOWN" - "BOTTOM UP" (of wel heen en weer) geeft echter mooie gelegenheid om deze contradicties tijdig op te sporen en correcties te plegen. Dit is erg handig, daar op verschillende niveaux BOTTOM en TOP van een procedure een andere definitie (inhoud) kunnen hebben. (De technicus ervaart een andere BOTTOM dan de astronoom.)

Welk inzicht men ook prefereert, één ding is uitermate belangrijk, namelijk de interface tussen de diverse niveaux. Veel spectaculaire fouten treden op, in grote informatica projecten, omdat de status van het systeem, verkregen op een hoger niveau, niet correct wordt doorgegeven aan routines op een meer elementair niveau. In FORTH hoeft men voor dit gevaar eigenlijk alleen de operationele stack in de gaten te houden, daar deze de verbinding tussen de routines vormt.
Verder moet men er wel op letten dat men de globale variabelen niet verandert in een subroutine tenzij wordt gegarandeerd dat deze variabele is hersteld zodra de routine wordt verlaten (bij hardware interrupts of catastrofes!).

Dit interface probleem is van kapitaal belang aangezien er meerdere mensen samenwerken in een gemeenschappelijk project. Het is dan ook het eerste punt dat moet worden bevroren, en gedurende de programma-ontwikkeling moet de specificatie streng worden nageleefd en gecontroleerd. Uiteindelijk zal er van elke routine een beschrijving moeten komen waaruit duidelijk blijkt wat er met de stack gebeurt, liefst een overzicht met de status van de stack voor en nadat de routine is gebruikt.
Verder is FORTH rijk aan namen en typen van variabelen, hetgeen gemakkelijk verwarring kan stichten, wat ook weer pleit voor een duidelijke beschrijving.

Opmerking :

- Terwijl wij tot nu toe over "BOTTOM UP" codering hebben gesproken wordt door de ontwikkelaars van gestructureerde programmering sterk aanbevolen om een "TOP DOWN" test van het systeem uit te voeren. Deze twee standpunten zijn - hoewel niet op het eerste gezicht - toch redelijk in overeenstemming met elkaar. "TOP DOWN" testen verwachten van de programmeur dat hij routines op meer elementair niveau vervangt door vereenvoudigde versies met dezelfde interface-eigenschappen. Deze manier van werken heeft wel een aantal voordelen, bijvoorbeeld:

  - een belangrijk programma kan al worden gebruikt (getest) voordat alle subroutines echt zijn geprogrammeerd.

  - de interfaces tussen niveaux kunnen al worden getest en er wordt tevens inzicht verkregen in de mogelijkheden om de interfaces compatible te houden.

Deze methode is dan ook geaccepteerd voor belangrijke projecten. Men neemt een routine op topniveau met een aantal routines van een dichtbijgelegen niveau en test deze in een enkele sessie op de computer. Toch wordt intern in zo'n verzameling van routines weer overgegaan op de "BOTTOM UP" methode, hetgeen overeenkomst met het FORTH concept niet alleen voor wat betreft de compilatietechniek maar ook wat betreft het testen en toevoegen aan het reeds bestaande pakket direct na compilatie. FORTH is nu een van de weinige systemen, die dit genre van faciliteiten geven ("incrementeel compileren"). FORTH procedures zijn over het algemeen erg kort en om deze reden is het absurd om "TOP DOWN" testen op alle niveaux consequent vast te willen houden.

Een voordeel van TOP DOWN" analyse is dat de specificaties van de interfaces tussen de niveaux, constructie van simulaties vergemakkelijkt en door FORTH kan, lang voordat het instrument is gerealiseerd, op hoog niveau dit logisch zeer goed worden aangepakt.

3.3 Compilatie en uitvoering

De compilatie van een procedure (beginnend met :) start met de constructie van een "bibliotheek-entry".
De pointer naar de voorgaande entry wordt pas ingevuld als de afsluiting (door ;) wordt gerealiseerd. Voorlopig bevindt dat adres zich nog in de locatie LAST .
STATE, die de compilayie of executie-mode bepaalt, wordt daarna gelijk gemaakt aan 1 (∅ = executiemode)
De tekst van de procedure wordt verder sequentiëel geanalyseerd, woord voor woord, onderwijl zoekend in de bibliotheek naar adressen welke corresponderen met de entry van het gezochte woord , en indien dit wordt gevonden, dan het adres van de pointer naar de geassocieerde code plaatsend op de volgende vrije plaats in de bibliotheek. Echter, als het "precedence bit" van het gevonden woord gelijk is aan 1 dan wordt niet het adres van de code in de bibliotheek geplaatst maar wordt die code uitgevoerd zonder enige vertraging.
Als uiteindelijk de ; wordt herkend (<;> speelt dezelfde rol als RETURN en END in Fortran) wordt STATE = ∅ (terug weer in executiemode) gemaakt en de "entry" is definitief in het systeem aangebracht, terwijl alle pointers bijgewerkt zijn.

Als een woord niet wordt gevonden stopt de compilatie en wordt controle gegeven aan de operateur. De compilatie moet dan weer opnieuw worden gestart na correctie.

Om tijdens de compilatie de executiemode te kunnen veranderen zijn er twee operatoren t.w.:

[stopt compilatie en start executie van de volgende woorden

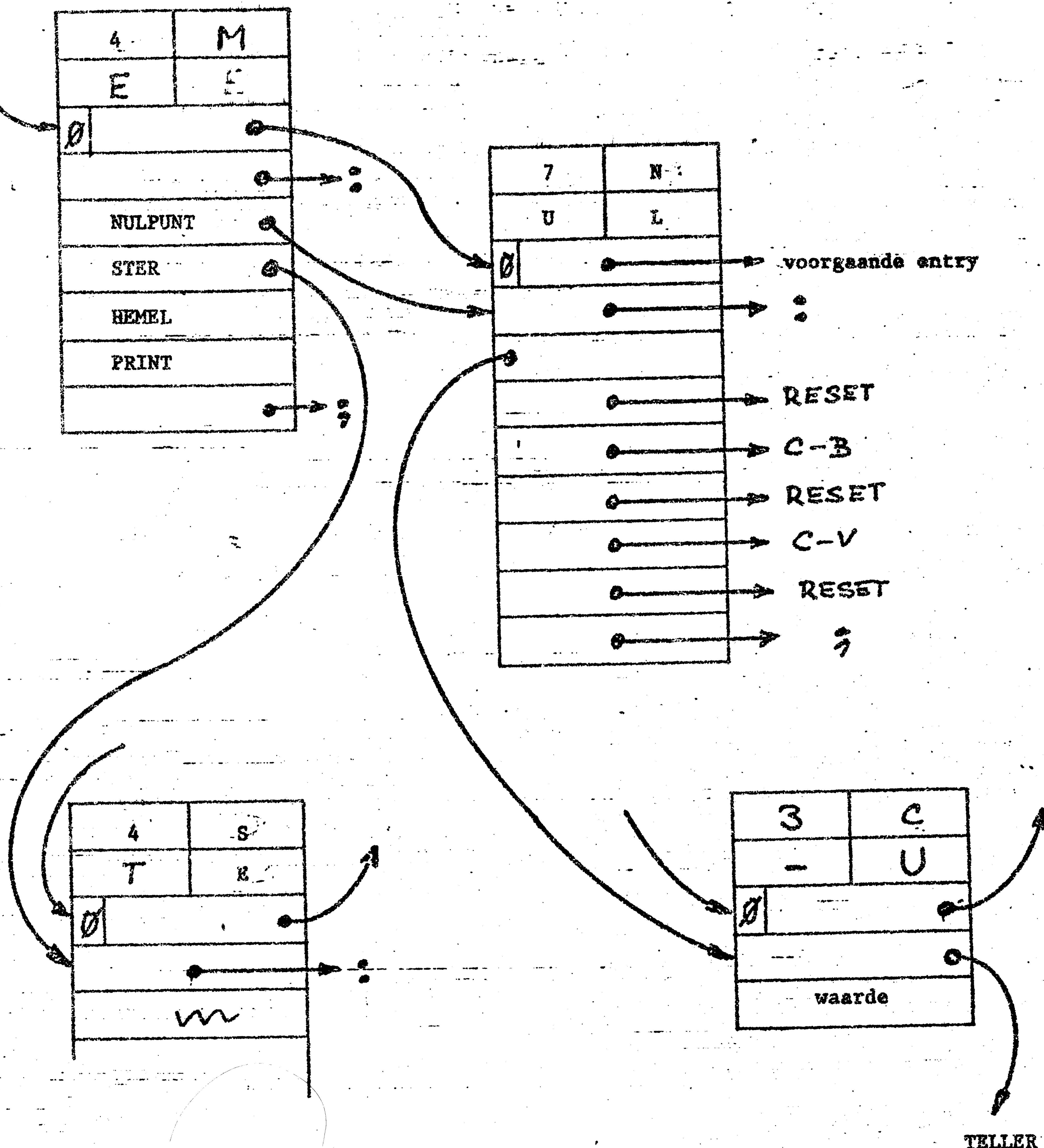
] stopt executie en start compilatie van de volgende woorden

Deze operatoren hebben verder geen invloed op de bibliotheek.

FORTH is bijzonder geschikt om compacte gecompliceerde routines te maken. In het algemeen kost de referentie naar een andere procedure maar één woord (op grote machines een half woord).

Er zijn enkele uitzonderingen op deze 1 op 1 relatie. Eén daarvan wordt besproken in § 4.6 . De andere betreft de "litterals" in een procedure.

GEHEUGENSTRUCTUUR NA COMPILATIE VAN HET VOORBEELD MEET (3.2.)



N.B. Een pointer is niets anders dan een woord dat het adres bevat van het objekt waarnaar wordt gepoint.

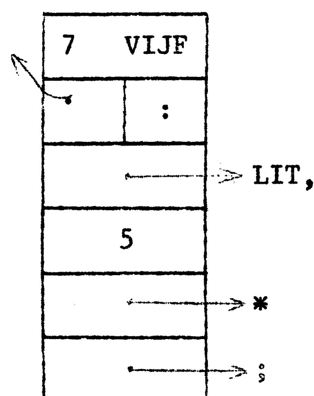
Neem bijvoorbeeld:

: VIJFVOUD 5 * ;

Deze routine vermenigvuldigt het getal op de stack met 5 .
De programmeur kan de volgende mogelijkheden kiezen:

1) Als de constante 5 erg veel voorkomt, dan kan er een entry worden gemaakt van het type constante met de waarde en de naam 5 . Op deze manier zijn ∅ en 1 reeds gedefinieerd in het basissysteem. Zodoende is er in VIJFVOUD een verwijzing naar een routine met de naam 5 die een vijf op de stack zet.

2) Zonder een definitie vooraf kan het ook. Als er niets wordt ondernomen dan zal de compiler automatisch bij het treffen van een cijfer (tijdens compilatie), in dit geval de 5, eerst een aanroep compileren naar de routine LIT , en daarachter het getal plaatsen. LIT, is een routine die het volgende getal op de stack plaatst.



<u>Executie</u>

Veronderstel dat wij de naam aanroepen (door hem bijv. in te typen) van een procedure of van een entry die al gecompileerd is.
Wat gebeurt er dan?

De interpreter vindt de entry in de bibliotheek en gaat, omdat ze in de executiemode is (en niet bezig met <:> --- <;>) direct over tot uitvoering van de geassocieerde code.

Zo kan men wel eenvoudig een 5 op de stack zetten door een 5 in te typen.

Nemen we het voorbeeld VIJFVOUD . De pointer naar de geassocieerde code wijst naar <:> welke een van de basisroutines van FORTH is. In de FORTH logica vormt de <:> een centrale beheerseenheid met de functie om alle volgende aangeduide codes achtereenvolgens uit te laten voeren. Dat betekent dat nu eerst de code wordt uitgevoerd die geassocieerd is met LIT, dan idem voor * en vervolgens voor <:> welke ieder voor zich mogelijk ook <:> routines zijn die de <:> routine voor hetzelfde doel gebruiken. Om al deze aanroepen te kunnen onderscheiden (voor <:>) is de instructie teller IC (FORTH instructie teller) in gebruik die altijd naar de volgende FORTH instructie wijst (een analoge rol als het P register van de processor).

Aan het eind van de executie van de routine LIT, moet deze , de instructie teller ophogen om de controle straks over te geven naar de volgende routine welke direct na 5 komt; dit is dus een extra ingreep (een extra increment om de 5 te passeren).

3.4 Recursieve aanroep - FORWARD - DEFINE

Recursieve aanroepen

De aanroep van een procedure binnen zichzelf kan niet in FORTH omdat de entry nog niet compleet is in de bibliotheek op het moment van aanroepen. Wel mag een synoniem worden gebruikt met dezelfde naam als de procedure zelf. Toch is dit erg handig, daar het mogelijk is om tamelijk complexe procedures op deze manier in twee fasen te realiseren zonder twee verschillende namen te moeten gebruiken. Dit is toegepast in het volgende voorbeeld (LIFE, Jan Vermue) : men heeft twee matrices OLD en NEW . EXCHANGE permuteert deze matrices. CLEAR plaatst Ø in de matrix NEW met:

```
: CLEAR CLEAR EXCHANGE CLEAR ;
```

Nu stopt men Ø in beide matrices.
In de nieuwe CLEAR wordt nu 2× de oude CLEAR aangeroepen, terwijl deze laatste vervolgens niet meer aanroepbaar is.

Het is mogelijk met behulp van de routine MYSELF een recursieve aanroep te genereren in FORTH . De procedure is als volgt gedefinieerd:

```
: MYSELF LAST @ , ; IMP MYSELF
```

Als MYSELF nu in een procedure gebruikt wordt is dit een recursieve manier om de actuele routine aan te roepen (zie het volgende voorbeeld).

Voorbeeld : Berekening van N ! = N. [(N-1)!] als N > 1
= 1 als N = 1 of Ø

```
: FACULTEIT DUP 2 < IF DROP 1
                    ELSE DUP 1- MYSELF *
                    THEN ;
```

Belangrijke opmerking :

- De return stack in FORTH geeft de mogelijkheid tot recursiviteit maar in FORTH worden de locale parameters en variabelen niet behandeld op deze stack. Dit is overgelaten aan de programmeur, die voor dit doel de operationele stack dan ook kan gebruiken. Hiervoor zijn enkele belangrijke regels:

1. Geen enkele variabele buiten die welke op de stack zijn geplaatst mag in een recursieve procedure worden veranderd.

2. Alle variabelen die veranderd mogen worden moeten worden gedupliceerd (worden geherdefinieerd) binnen in de recursieve handeling.

3. Alle parameters moeten geherdefinieerd zijn, voordat de recursieve procedure aanroep gepleegd wordt (MYSELF).

Men moet op de begrenzing van de stacks passen, vooral op die van de return stack:

Zij r het aantal recursieve aanroepen
m het aantal parameters
n het aantal locale variabelen
d het aantal stappen in de DO loop in een recursie stap,

dan zijn

r (m + n) posities op de operationele stack
r (1 + kd) posities op de return stack

nodig, waarin

k het aantal posities per DO-loop stap (d.i. 2 of 3) is.

Afgezien van de lengte van de stacks geven recursieve procedures ook nog vaak aanleiding tot de volgende fouten:

- locale parameters en variabelen zijn niet op het goede moment op de stack of er wordt vergeten ze op de stack te plaatsen aan het eind van de procedure;

- de diepte van de recursiviteit is niet voldoende beheerst met het risico dat door "geleveled" wordt tot ongedefinieerdheid.

Het volgende geval tenslotte is een lineaire hiërarchieke constructie die niet mogelijk is:

```
: A B ;
: B A ;
```

Met de volgende kunstgreep is dit echter op te lossen:

```
FORWARD   A
:  B      A  ;
DEFINE    A  :  A  B  ;
```

FORWARD A maakt een bibliotheek entry voor A die door B kan worden gebruikt. DEFINE A modificeert deze dummy entry volgens de procedure die direct hierna wordt gegeven, vervangt de dummy door de bedoelde procedure. Deze manier kan ook worden gebruikt om recursiviтiet te maken, maar is trager door een extra aanroep in het eerder gegeven voorbeeld.

De voorbeelden in V.3 en V.4 gebruiken de recursiviteit en de voorwaartse definitie.

Het geval van FACULTEIT kan best worden opgelost zonder recursiviteit, VON-KOCH en HILBERT daarentegen zijn bijna niet te programmeren zonder deze mogelijkheden (zie V.3 en V.4).

N.B.: Er is geen beperking in het aantal procedures gedefinieerd met FORWARD < naam >

- Men mag net zoveel definities geven als men wil tussen FORWARD < naam > en DEFINE < naam >
- DEFINE < naam > moet direct voor de corresponderende constructie komen. Strikt genomen mag er niets tussen de twee in staan.
- Het is niet absoluut noodzakelijk dat de naam van de constructie dezelfde is als die van DEFINE. Daarentegen moeten de namen volgend op FORWARD en DEFINE identiek zijn voor eenzelfde procedure.

3.5 Operatoren werkend op de bibliotheek (woordenlijst) (gedurende de compilatie)

In principe zijn alle bewerkingen op de woordenlijst automatisch. Het kan echter voorkomen dat men de bibliotheek wil manipuleren om bijvoorbeeld nieuwe procedures voor de interpreter te maken.

Hiervoor zijn de volgende operatoren:

HERE — zet het adres van het eerste vrije woord in de bibliotheek op de stack.

, — zet de inhoud van de stack in de plaats aangewezen door HERE, terwijl HERE met 1 wordt verhoogd. (Precedence bit = Ø).

ADOPT — als < , > maar nu met precedence bit = 1 , d.w.z. hij wordt uitgevoerd tijdens compilatie en niet tijdens executie fase.

DP — variabele (of register) die het adres van het eerste vrije woord in de bibliohteek aangeeft.

Als DP een register is, dan is hij niet anders aanroepbaar dan met

DP*a* DP! en DP+!

Voorbeeld : HERE is gecodeerd als : HERE DP *a* ;
of : HERE DP*a* ;

ØIMP < naam > — verandert het precedence bit van de routine naar Ø .

1IMP < naam > — idem, precedence bit naar 1 .

?DEF < naam > — laat een :
1 op de stack als de procedure al een synoniem heeft ;
Ø op de stack als de naam not niet is gebruikt .

COMPILE < naam > — zet in de volgende plaats van de bibliotheek het adres van de pointer die wijst naar de code behorend bij < naam > onafhankelijk van de toestand van het precedence bit.

FORGET < naam > vergeet alle definities in de bibliotheek na en inclusief de aangehaalde procedure (naam). Zijn er synoniemen in het spel, dan wordt alles tot en met het laatst gedefinieerde synoniem vergeten. Op deze manier wordt het voorgaande aan de synoniem weer bereikbaar.

Opmerking :

- FORTH start altijd met een lege definitie : TASK ;
  Door FORGET TASK te geven, kan alles wat zelf is gedefinieerd weer worden vergeten.
  Men kan deze techniek zelf erg handig gebruiken tijdens programma-ontwikkeling.

3.6 Wezen en hun adoptie

Wezen zijn procedures (normaal of in assembler) die slechts voor een zeer specifiek geval worden gebruikt. De header, in gebruik om terugvinden mogelijk te maken, is in dit geval niet nodig en alleen een pointer naar de geassocieerde code is noodzakelijk.

Er zijn twee FORTH-manieren om wezen te maken:
- met de procedures ORPHAN en :ORPHAN
- met de procedure ORCX of :ORX

Procedures ORPHAN (Assembler) en :ORPHAN (Forth)

Deze procedures creëren een gereduceerde entry (alleen pointer naar geassocieerde procedure) en laten het adres van deze pointer op de operationele stack.

Vervolgens kan de pointer worden opgepakt met de operator ADOPT binnen een normale procedure.

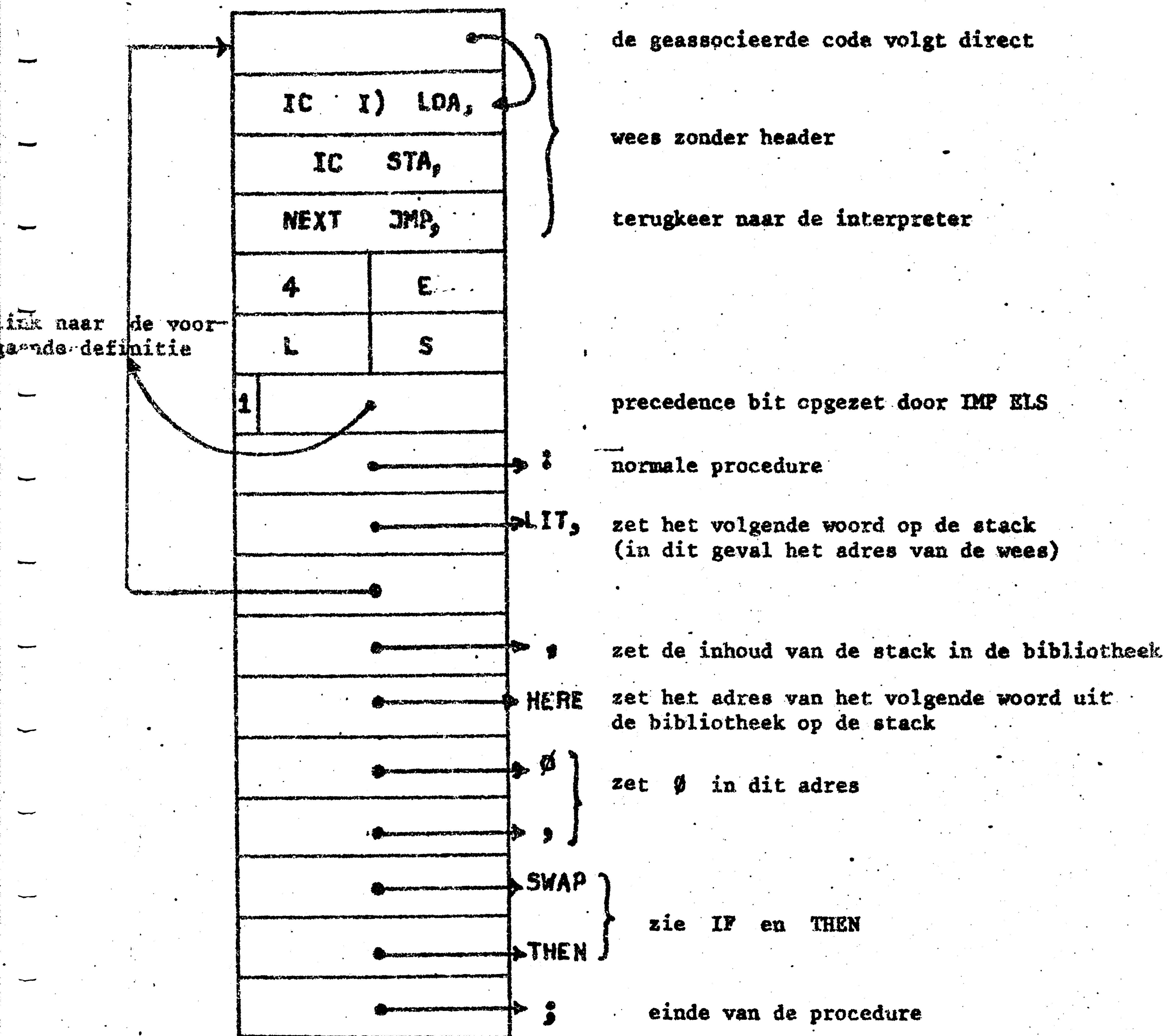
Op deze manier is de wees ondergebracht.

Voorbeeld : De operator ELSE voor de HP 2100 versie

```
ORPHAN (Spring naar de inhoud van het volgende woord in
       de procedure)
       IC I) LDA,  IC STA,  NEXT,

:  ELSE   LIT,   ADOPT   ,   HERE Ø ,   SWAP THEN ;
IMP ELSE
```

Na compilatie wordt de volgende structuur in het geheugen gevonden.


IC I) LDA,
IC STA,
NEXT JMP,
4
E
L
S
1
link naar de voorgaande definitie
LIT,
HERE
Ø
SWAP
THEN
de geassocieerde code volgt direct
wees zonder header
terugkeer naar de interpreter
precedence bit opgezet door IMP ELS
normale procedure
zet het volgende woord op de stack
(in dit geval het adres van de wees)
zet de inhoud van de stack in de bibliotheek
zet het adres van het volgende woord uit
de bibliotheek op de stack
zet Ø in dit adres
zie IF en THEN
einde van de procedure

:ORPHAN doet hetzelfde maar nu voor een FORTH procedure.

ORCx en :ORx met x*) is een cijfer (0 → 9) spelen dezelfde rol als ORPHAN en :ORPHAN maar maken het mogelijk verschillende wezen tegelijk te maken (10 stuks). In dit geval echter wordt het adres van de wees niet op de stack gezet, maar in een buffertje geplaatst. De wees wordt geadopteerd met ADOx waarin x correspondeert met de gewenste wees.
Dit adres blijft beschikbaar totdat de creatie van een nieuwe wees er over heen schrijft.

Men moet erg oppassen met re-entrant procedures, daar het mechanisme met de wezen hierop (nog) niet berekend is.

*) of A → N met A is synoniem voor 1
B is synoniem voor 2
etc.

3.7 Drie fundamentele procedures

NEXT : en ;

Deze procedures zijn van vitaal belang voor de werking van FORTH. Zij maken het mogelijk om de controle over te geven aan simultane en sequentiële procedures. Daar zij erg vaak worden gebruikt, moeten zij met betrekking tot de gebruikte processor geoptimaliseerd zijn.

Onderaan in de hiërarchie van een procedure, d.w.z. de procedure die zelf geen andere meer aanroept, vindt men een in machinetaal geschreven routine. Deze routine moet impliciet of expliciet (zie 5.3) eindigen met NEXT . NEXT maakt het mogelijk om controle over te geven naar een andere routine op hetzelfde niveau. Merk op dat een in assembler geschreven routine ook dient te eindigen met NEXT .

: en ; openen en sluiten een routine op hoog niveau (alles behalve assembler). Deze procedures beheersen de return stack.

De volgende voorbeelden beschrijven de implementatie van deze procedures op een HP 2100 processor. Dit is een machine met twee arithmetische (algemene) registers A en B die ook als gewone adressen (Ø en 1) behandeld mogen worden.

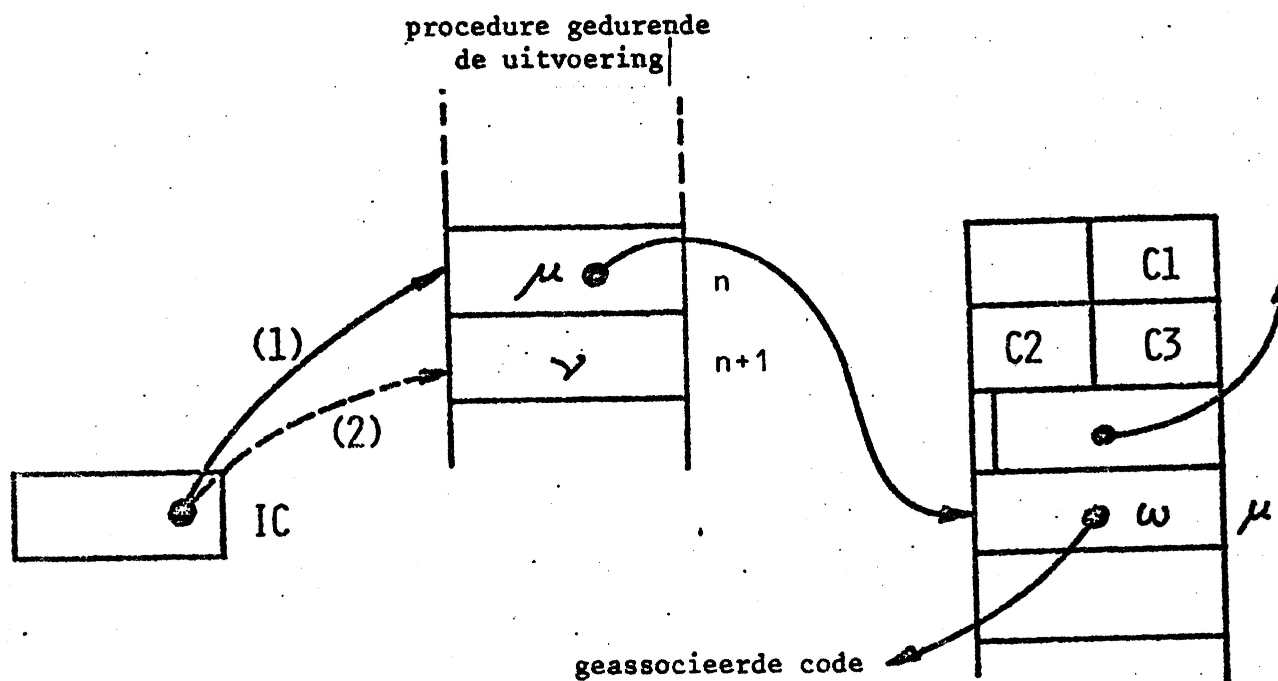
LDA, LDB	breng de inhoud van de geheugenplaats over naar het gerefereerde register,
STA, STB	plaats de inhoud van het register in de geheugenplaats,
ISZ	increment de geheugenplaats en skip als Ø ,
INA, INB	tel 1 op bij de inhoud van het register,
-INR	een macro die één aftrekt in een geheugen locatie,
JMP	een onconditionele sprong.

1) Verbindingen met de strakke lijn geven de status vóór uitvoering van de code.

2) Met stippellijn geven ze de status nà uitvoering

Kleine letters geven geheugenadressen.
Staan ze buiten een vak, dan is het het adres van dat vak.
Staan ze er in, dan is het een pointer naar een adres.

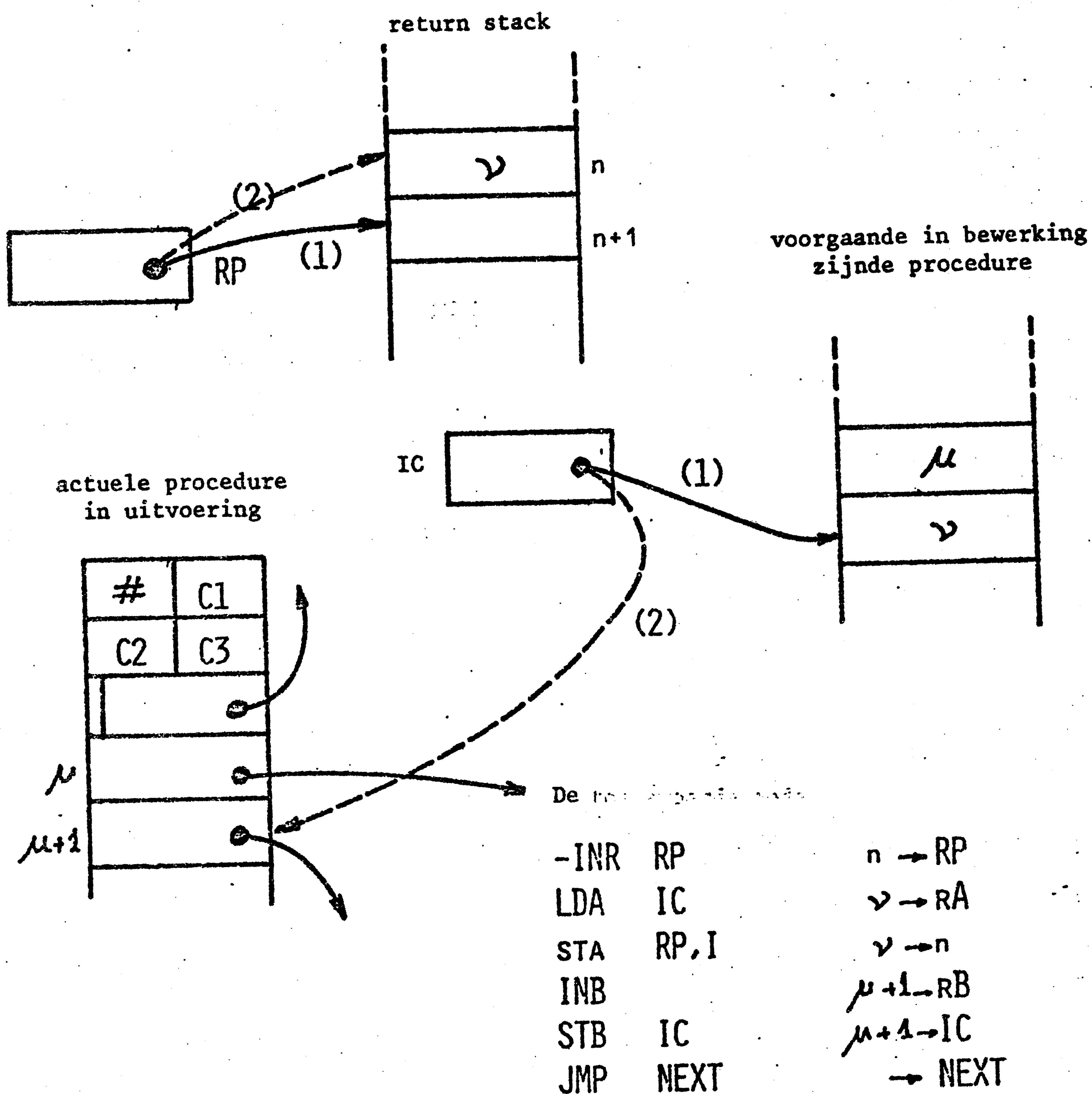
Uitvoering van NEXT



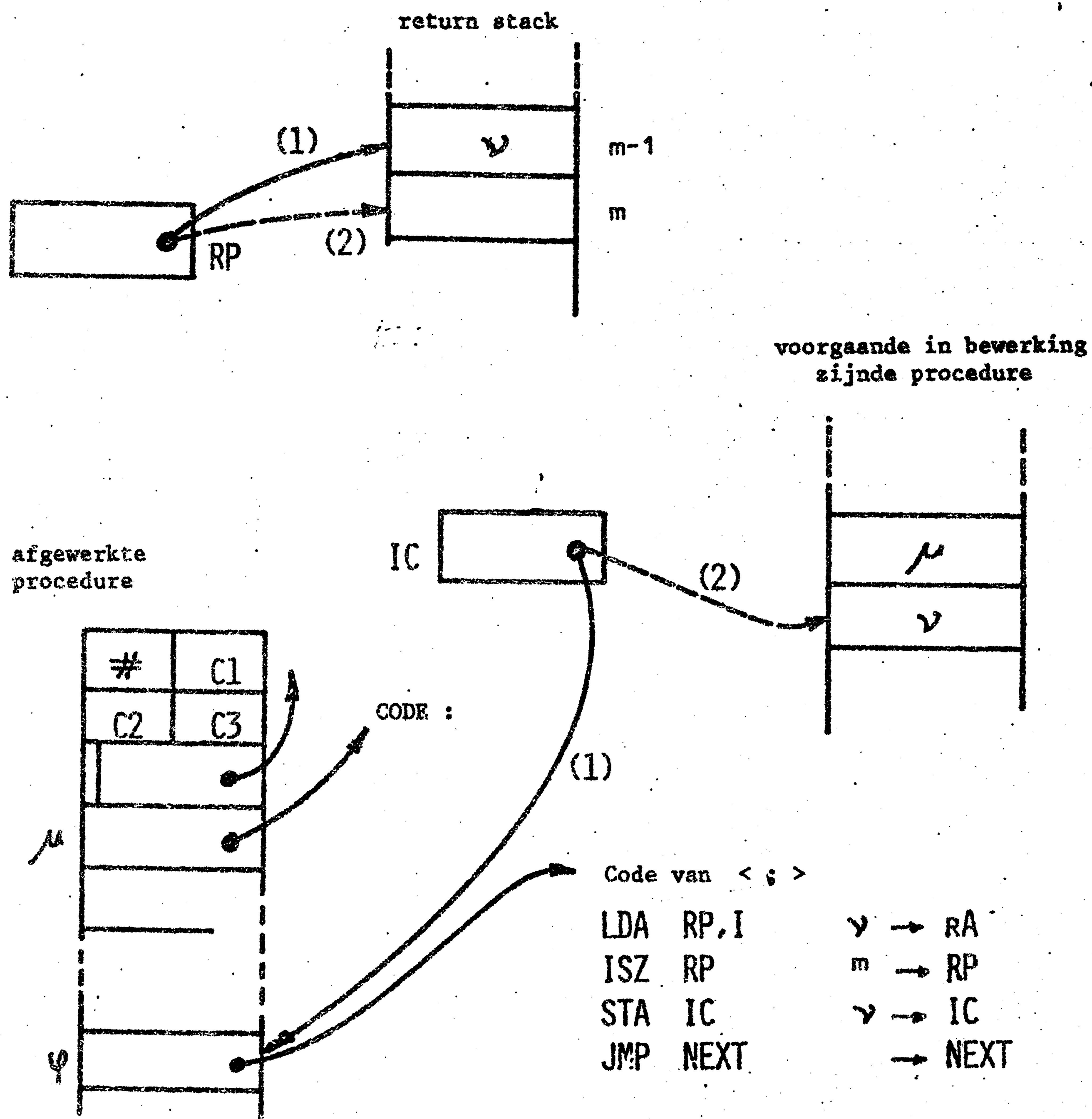

```
NEXT   LDB   IC,I      μ   → RB
       ISZ   IC        n+1 → IC
.NEXT  LDA   B,I       ω   → RA
       JMP   A,I           → ω
```

IC wijst altijd naar de volgende procedure
die moet worden uitgevoerd

Uitvoering van de met < : > geassocieerde code



Uitvoering van de met < ; > geassocieerde code



3.8 De vocabulaires

Zoals men in het begin van dit hoofdstuk kan zien kan de bibliotheek een boomstruktuur van "lijsten" zijn waarin het mogelijk is gebruik te maken van diverse deelverzamelingen (takken).

Deze vertakte structuur is alleen een logische weergave waar niets van te zien is in het geheugen. De entries van de vocabulaires ASSEMBLER en FORTH bijvoorbeeld staan in het geheugen gewoon door elkaar.

Een vocabulaire wordt gecreëerd met de volgende procedure (onafhankelijk van de inhoud van deze vocabulaire):

VOCABULARY < naam >

Direct na het geven van deze opdracht bevindt men zich in de nieuwe vocabulaire (de actuele ~).

Opmerking

- Noch CONTEXT noch CURRENT zijn op dit moment veranderd.

Een vocabulaire kan op twee wèl verschillende manieren worden gebruikt:

1. bij het opzoeken van een reeds gecreëerde entry,
2. bij het toevoegen van een entry aan de lijst.

Elke vocabulaire heeft een pointer die naar de laatste toegevoegde entry wijst. Twee pointers in het systeem geven elk ogenblik aan

- in welke vocabulaire er gezocht zal worden : CONTEXT ,
- in welke vocabulaire de volgende entry zal worden geplaatst : CURRENT .

CONTEXT en CURRENT wijzen in feite naar een pointer van de bibliotheek die zelf naar de laatste entry van deze bibliotheek wijst. Zo geven CONTEXT of CURRENT het adres van de laatste entry van de corresponderende vocabulaire.

Men krijgt toegang tot een vocabulaire door simpelweg zijn naam te geven, waardoor de waarde van CONTEXT wordt veranderd. Er is maar één vocabulaire tegelijk aanroepbaar; ondanks dat kunnen de verschillende vocabulaires aan elkaar "bedraad" worden, waardoor wel gezocht naar niets veranderd kan worden in een andere vocabulaire. Het is geen probleem synoniemen te definiëren in verschillende vocabulaires.

Om aan een vocabulaire een nieuwe definitie toe te voegen moet men het commando:

< naam vocabulaire > DEFINITIONS

geven. Hiermede wordt

CURRENT

veranderd en kunnen definities aan de vocabulaire worden toegevoegd.

Opmerkingen :

- Het precedence bit van de vocabulaire is 1 en verandering van vocabulaire kan binnen in een procedure gedurende de compilatie worden veranderd.
- Er treedt een automatische verandering van vocabulaire op wanneer men het volgende gebruikt:

  : :ORx en ;: die dwingen tot zoeken in de bibliotheek waar de nieuwe definitie geplaatst gaat worden.

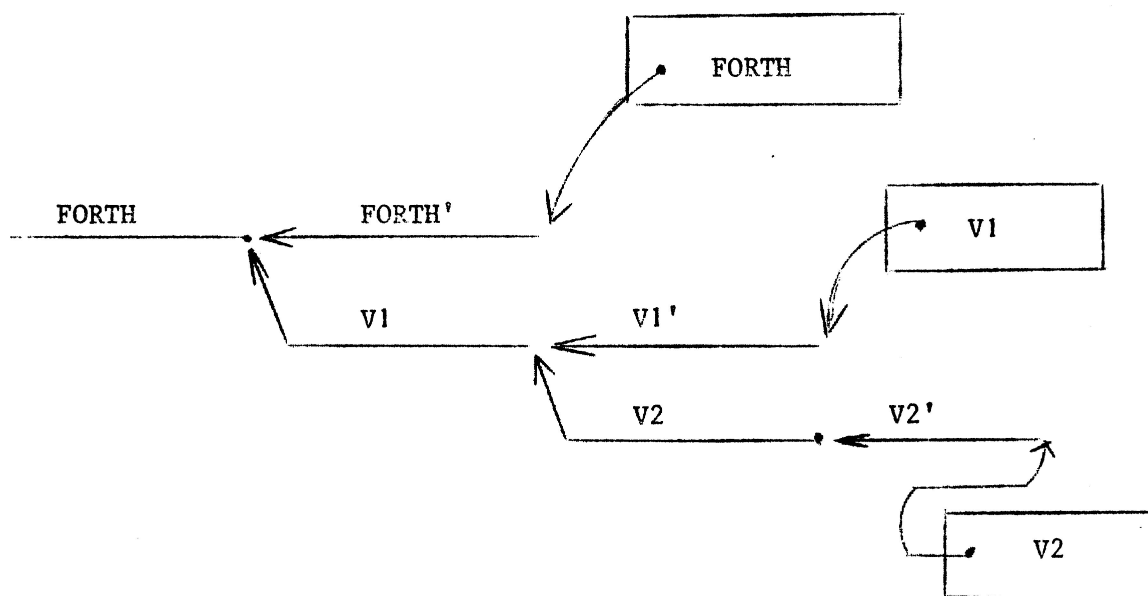
  CODE , ,CODE en ;CODE dwingen tot zoeken in de vocabulaire ASSEMBLER welke aan de FORTH vocabulaire is bedraad.

In onderstaande afbeelding wordt de boomstructuur wat beter uitgewerkt.

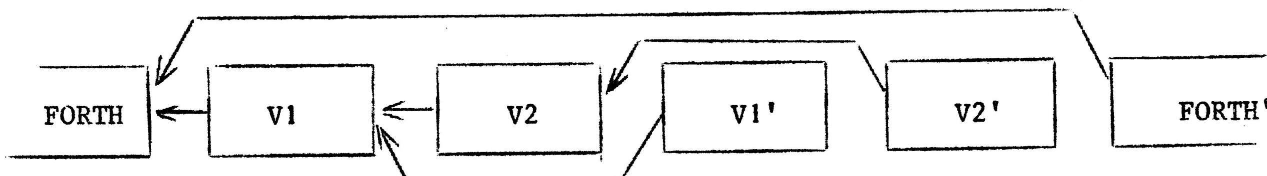
Veronderstel dat we de volgende constructie hebben:

1. FORTH DEFINITIONS
2. VOCABULARY V1 V1 DEFINITIONS < entries voor V1 >
3. VOCABULARY V2 V2 DEFINITIONS < entries voor V2 >
4. V1 DEFINITIONS < entries voor V1' >
5. V2 DEFINITIONS < entries voor V2' >
6. FORTH DEFINITIONS < entries voor FORTH' >

De bibliotheek heeft de volgende logische constructie:



Als we in het geheugen kijken, dan zien we de volgende structuur (in volgorde van programmering):



De entries van FORTH' zijn niet bereikbaar voor de andere vocabulaires.

De entries van V1 zijn bereikbaar door V1' , V2 en V2' , maar niet door FORTH' .

De entries van V1' , V2 en V2' zijn niet bereikbaar voor de rest.

Met het commando CHAIN kunnen we automatisch vocabulaires aan elkaar koppelen.
Het gebruik is als volgt:
Na de actuele vocabulaire gedefinieerd te hebben wordt gegeven

CHAIN < naam 2de vocabulaire >

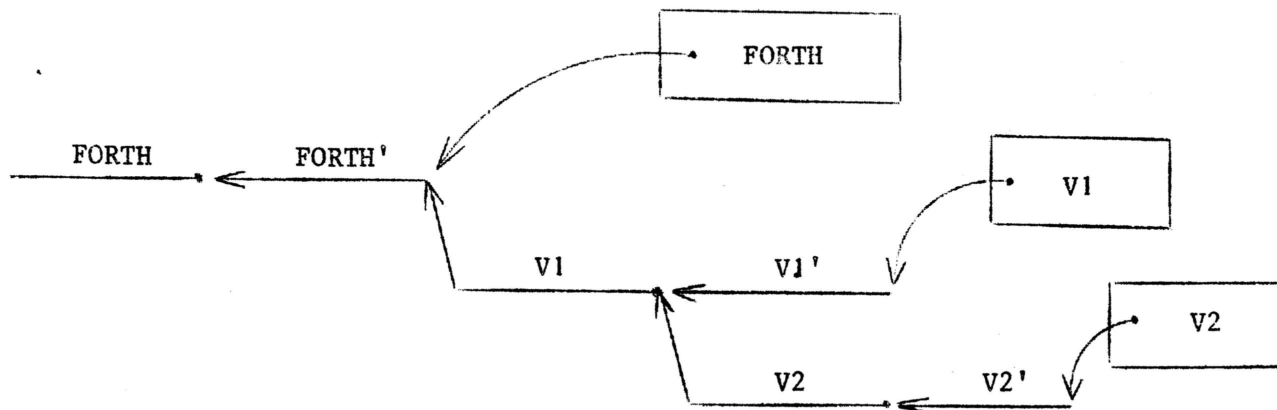
Bij voorbeeld, als men regel 2 van het voorgaande voorbeeld verandert tot:

VOCABULARY V1 V1 DEFINITIONS CHAIN FORTH

< entries voor V1 >

dan zullen alle toevoegingen FORTH' een uitbreiding volgend op de vocabulaire FORTH toegankelijk zijn voor V1 , V1' , V2 en V2' .

Men krijgt nu de volgende logische structuur:



Opmerkingen :

- FORTH' , V1' en V2' zijn dialecten van FORTH , V1 en V2 ,
- het actuele dialect is de niet onderbroken keten waarin nu de nieuwe entries worden gemaakt.

Het is duidelijk dat het commando FORGET alleen mag worden gebruikt om in het actuele dialect entries weg te halen. De programmeur moet zich hiervan verzekeren daar hij anders onherstelbare schade aanricht aan zijn systeem.

4.0 Lussen en voorwaardelijke sprongen

FORTH kent noch LABELS , noch GOTO statements. Herhalingen en keuze-sprongen gebeuren met enkele eenvoudige stuuroperatoren. Hiermee is het mogelijk vrijwel alle voorkomende gevallen op elegante wijze op te lossen.

Gekoppeld aan een hiërarchisch opgebouwde structuur van procedures leidt dit tot de volgende kenmerken :

- elk blok van een organigram heeft één in- en één uitgang,
- uitvoering geschiedt altijd van boven naar beneden en van links naar rechts (per regel).

In FORTH zijn de volgende voorwaardelijke sprong- en lus-operatoren gedefinieerd :

1. Voorwaardelijke sprong met enkelvoudige keuze (4.2)

```
V   IF   P    THEN
V   IF   P1   ELSE   P2   THEN
```

waar / onwaar

THEN } sluit
ESAC } woorden
haalt N van stack

2. Voorwaardelijke sprong met meervoudige keuze (4.3)

```
N   N1  CASE  P1   ELSE   N2  CASE  P2  ...   ESAC
```

THEN THEN THEN

↑ mag ook procedure zijn, die getal op stek achterlaat.

3. Herhalingslus met index (4.4)

```
N1   N2   DO   P        LOOP
N1   N2   DO   P   M   +LOOP
```

↑ eind+1 ↑ begin

4. Lus met voorwaardelijke sprong (4.5)

```
BEGIN   P    V   END
LOOP:   P1   V   WHILE   P2   REPEAT
```

~~LOOP:~~ BEGIN

N1 N2 ... M zijn gehele getallen
V is een voorwaarde
P , P1 en P2 zijn procedure-lichamen.

Voor iemand die FORTRAN gewend is lijkt dit een willekeurige, beperkte set van operatoren. Echter, men kan aantonen dat bovenstaande set van operatoren reeds meer bevat dan minimaal noodzakelijk is voor het oplossen van alle voorkomende gevallen en overeenkomt met die van de meest recente talen als PASCAL en ALGOL 68.

N1 = 100 dan laatste I = 99

Ter verduidelijking :

In FORTRAN worden de verschillende programmadelen aangegeven door labels. Het uitvoeren van een programma geschiedt door sprong-opdrachten (GOTO) naar deze labels (al dan niet expliciet, zoals in een IF statement). Met deze label is het mogelijk kris-kras door het programma te springen met alle mogelijke gevolgen van dien.

In FORTH (en in ALGOL 68 en in PASCAL bijv.) bakenen de operatoren BEGIN ... END , DO ... LOOP , IF ... ELSE ... THEN duidelijk een programmagebied af. Het is niet mogelijk dit gebied op een andere manier binnen te komen. Op dezelfde éénduidige wijze is gedefinieerd waar het gebied wordt verlaten. Er zijn twee uitzonderingen op deze regel: de procedure EXITLOOP maakt het mogelijk uit een DO ... LOOP te komen, de procedure ABORT stopt het lopende programma en geeft controle terug aan de operator (fout situatie).

4.1 De vergelijkingsoperatoren

De vergelijkingsoperatoren vervangen de operand(en) op de stack door het resultaat van de vergelijking (≠Ø is TRUE of =Ø is FALSE). Deze operatoren nemen dezelfde prefix (D, F) als de rekenkundige operatoren. Het resultaat is altijd een enkelvoudig geheel getal (Ø of ≠Ø) .

Bij operatoren met twee operanden wordt de relatie van links naar rechts gelezen, evenals dit het geval is bij de rekenkundige bewerkingen.

vb. 4 3 < geeft Ø (4 < 3 is FALSE) .

Beschikbaar zijn de operatoren:

n Ø=	test of het getal op de stack nul is,
n Ø<	test of het getal op de stack negatief is,
n Ø≠	test of het getal op de stack niet nul is,
n m =	test de gelijkheid (n = m) ,
n m <	test de ongelijkheid (n < m) ,
n m >	test de ongelijkheid (n > m) .

Zie ook in 2.3

4.2 Enkelvoudige voorwaardelijke sprong

De operatoren IF en WHILE testen de top van de stack (en vernietigen deze waarden, evenals alle operatoren doen) en

- gaan door met de daaropvolgende procedure als de top ongelijk nul is ,
- springt indien hij nul is naar de procedure volgend op ELSE , of volgend op THEN (als ELSE afwezig is) , of volgend op REPEAT bij een zg. voorwaardelijke lus.

na positieve IF De operator ELSE veroorzaakt een onvoorwaardelijke sprong naar de procedure volgend op THEN (slaat alles tussen ELSE ... THEN over). Hij is niet in alle gevallen noodzakelijk en mag niet worden gebruikt in een voorwaardelijke lus.

De operator THEN geeft het einde van een voorwaardelijk blok aan en is dus absoluut noodzakelijk.

IF en THEN kunnen worden beschouwd als twee sluithaakjes om een zin.

Noot : - Het is niet nodig dat de voorwaarde net vóór de IF wordt berekend. Hij kan op de stack achtergelaten zijn door elke willekeurige andere procedure.

Voorbeelden: - Men wil een regel { spatiëren / opvoeren } (line feed) als

CONTROL = 1Ø

CONTROL 10 = IF CR THEN

- Men wil van een even getal zijn kwadraat en de derde macht van een oneven getal afdrukken.

```
: C2C3 DUP 2 MOD IF DUP DUP * * ELSE DUP * THEN . ;
```

4.3 Voorwaardelijke sprong met meervoudige keuze

Voor deze constructie is vooralsnog geen eenduidige afspraak gemaakt, bij de theoretici, noch bij de gebruikers. Hij is bekend onder de namen computed GOTO , CASEOF , CASE , SELECT etc. met evenveel varianten als namen.

De hier gebruikte variant komt overeen met CASE van PASCAL : de waarde op de stack wordt vergeleken met een lijst van vooraf vastgestelde waarden. Zodra een overeenstemming is bereikt, wordt het daarbij behorende procedurelichaam uitgevoerd.

In FORTH vergelijkt CASE de twee waarden op de top van de stack.

- Als zij aan elkaar gelijk zijn, dan worden de beide waarden vernietigd en dan wordt de uitdrukking uitgevoerd tot aan de eerste ELSE . Hierna wordt uit de constructie gesprongen.
- Als zij verschillend zijn, wordt enkel de top van de stack vernietigd en dan gaat de uitvoering over naar de uitdrukking volgend op de eerstvolgende ELSE .
- CASE kan worden gebruikt voor het vervangen zowel van een IF met tweevoudige vertakking als van een "IF met meervoudige vertakkingen".

Het volgende voorbeeld licht dit toe :

Als BASE het getal 8_{10} bevat, druk dan OCTAL af
Zo niet , druk dan OTHER af .

```
: BASE? @ 8 CASE ." OCTAL"
            ELSE ." OTHER" DROP
            THEN ;
```

Voeg hierbij de gevallen van 10-tallige (decimale) en 16-tallige (hexadecimale) stelsels :

```
: BASE? BASE @          8  CASE ." OCTAL"
                ELSE   10  CASE ." DECIMAL"
                ELSE   16  CASE ." HEXADECIMAL"
                ELSE DECIMAL  DUP . ." BASE 10" BASE !
                           THEN THEN THEN ;
```

Opmerkingen :

- Tenminste één ELSE moet worden gebruikt voor het geval dat de gelijkheid niet optreedt.
- De constructie moet eindigen met evenveel THEN's als er ELSE's waren.
- Op enkele installaties vervangt ESAC alle THEN's hoeveel er ook maar mogen zijn (ESAC is de omkering van CASE). Hier niet.

4.4 Lussen (DO-LOOP's) met index

De DO operator in FORTH is vrijwel gelijk aan de DO in FORTRAN, echter met enkele kleine verschillen.

De DO operator moet altijd op de stack vinden:
de beginwaarde van de index op de top
de eindwaarde daar net onder.

Beide getallen moeten heel zijn, maar kunnen verder positief, negatief of nul zijn.

De operator LOOP hoogt de index met 1 op, vergelijkt het resultaat met de eindwaarde en herhaalt zonodig de procedure volgend op DO .

De operator +LOOP als LOOP , maar hoogt de index op met de waarde op de stack (positief, negatief of nul).

bestaan niet

De operator BACKLOOP verlaagt de index met 1 .

De operator -LOOP verlaagt de index met de waarde op de stack.

De operator EXITLOOP maakt het mogelijk om uit de lus te springen, onafhankelijk van de waarde van de index.

LEAVE

Wat betreft het bereiken van de eindwaarde:
De lus wordt verlaten nadat de index is geïncrementeerd als
- de eindwaarde bereikt is èn het increment positief is;
- de eindwaarde overschreden is èn het increment negatief is.

Opmerkingen :

- De operatoren DO en LOOP (of +LOOP etc.) vormen de twee haakjes om een herhalingsblok en dienen in één en dezelfde procedure te worden gebruikt. De lus-index zelf is alleen toegankelijk binnen in de procedure die door DO en LOOP wordt omsloten.
- Verscheidene DO ... LOOP lussen mogen gerust dakpansgewijs over elkaar worden gelegd (binnen in één en dezelfde procedure).
- De operator I zet de index op de stack van de - op het tijdstip van aanroep - meest naar binnen gelegen lus.
- De operator J zet in de net hierboven genoemde lus de index op de stack van de lus die één niveau meer naar buiten ligt.

- De operator K evenzo voor de lus die twee niveaux meer naar buiten ligt.

- Er is in het algemeen geen operator gedefinieerd voor indices van nog verder naar buiten liggende lussen.

- Een DO lus wordt altijd tenminste één keer uitgevoerd.

- Het increment van de +LOOP operator kan tussen lus-uitvoeringen worden gewijzigd. Evenals alle FORTH operatoren vernietigt +LOOP zijn operand.

Voorbeeld -

- Druk de getallen van Ø tot en met 1Ø af

```
... 11 Ø DO I . LOOP ...
```

- In afnemende volgorde

```
Ø 1Ø DO I . -1 +LOOP
```

- Berekening van N! :

```
(N) : N! 1 SWAP 1+ 1 DO I * LOOP ;
```

- Het afdrukken van een vermenigvuldigingstabel in de actuele getallenbasis (BASE) :

```
BASE @ OCTAL
:"*TITLE ." MULTIPLICATION TABLE IN BASE " BASE 0
          10 CASE ." OCTAL"
     ELSE 12 CASE ." DECIMAL"
     ELSE 20 CASE ." HEXADECIMAL"
     ELSE          0. ." (BASE 8) "  THEN THEN THEN ;
: *TABLE 3 FLD ! CR CR CR *TITLE CR CR
        BASE @ DUP 1 DO CR .
                    DUP 1 DO
                           I J * S.
                        LOOP
                  LOOP
        DROP CR CR ;    BASE !        ;S
```

- Het afdrukken van de inhoud van de operationele stack (de top van de stack bevindt zich rechts)

```
1Ø CYCLE ICR
: SDUMP  1 ' ICR !  7 FLD !  CR
        SP @ 1+ SPØ @ 1- DO
                          I @ S.
                          ICR Ø= IF CR THEN
                     -1 +LOOP  CR ;
```

Opmerking :

- In dit voorbeeld wordt een variabele van het type CYCLE gebruikt voor het regelen van de afdruk.
  Merk op dat SDUMP tenminste één waarde zal afdrukken - ook al zou de stack leeg zijn, op het moment van aanroep van de procedure SDUMP.

Oefening :

- Maak zelf een nieuwe definitie voor SDUMP zodat er geen waarde wordt afgedrukt als de stack inderdaad leeg zou zijn.

OCTAL *TABLE   HEX *TABLE   5 BASE !   *TABLE

MULTIPLICATION TABLE in BASE OCTAL

1	2	3	4	5	6	7
2	4	6	1Ø	12	14	16
3	6	11	14	17	22	25
4	1Ø	14	2Ø	24	3Ø	34
5	12	17	24	31	36	43
6	14	22	3Ø	36	44	52
7	16	25	34	43	52	61

MULTIPLICATION TABLE in BASE HEXADECIMAL

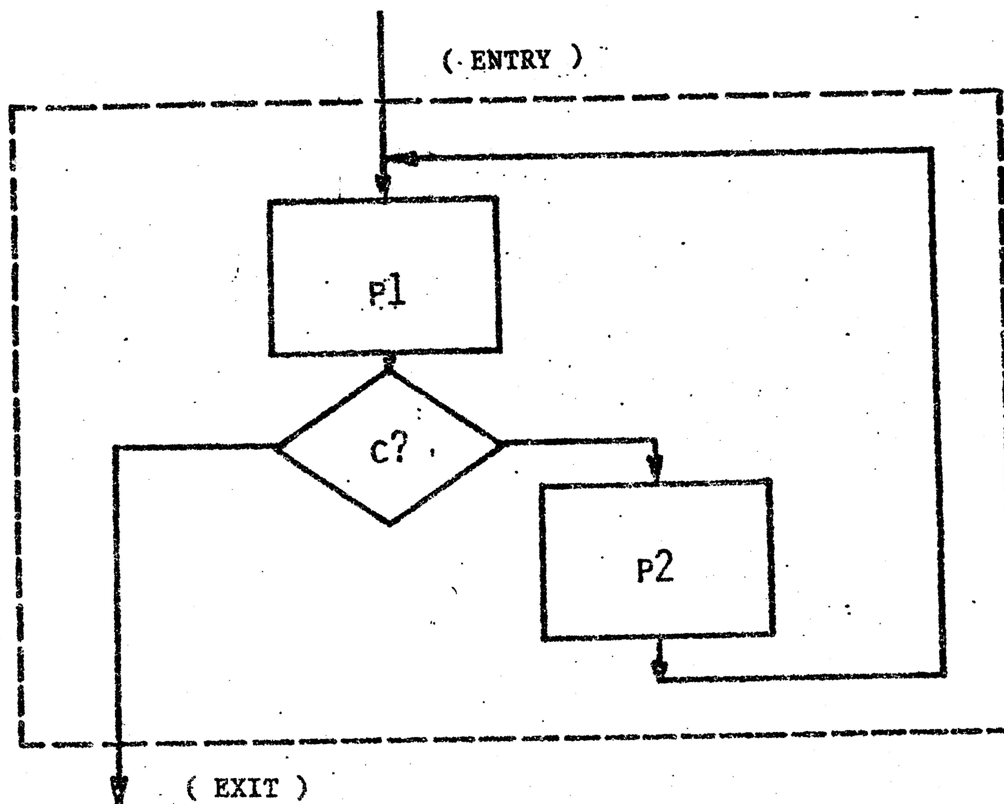
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	4	6	8	A	C	E	1Ø	12	14	16	18	1A	1C	1E
3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	8	C	1Ø	14	18	1C	2Ø	24	28	2C	3Ø	34	38	3C
5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	C	12	18	1E	24	2A	3Ø	36	3C	42	48	4E	54	5A
7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	1Ø	18	2Ø	28	3Ø	38	4Ø	48	5Ø	58	6Ø	68	7Ø	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	14	1E	28	32	3C	46	5Ø	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	3Ø	3C	48	54	6Ø	6C	78	84	9Ø	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	7Ø	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

MULTIPLICATION TABLE in BASE 5 (BASE 8)

1	2	3	4
2	4	11	13
3	11	14	22
4	13	22	31

4.5 Voorwaardelijke (BEGIN, END, WHILE) lussen .

Bekijk het volgende organigram:



Na binnenkomen in het blok voeren we eerst P1 uit en dan - afhankelijk van de waarde op de stack - voeren we P2 uit.

Het is opmerkelijk dat talen gegroeid uit ALGOL 60 , zoals PASCAL , - hoewel zij goed voorzien zijn voor wat betreft voorwaardelijke lussen - het bovenstaande toch niet op eenvoudige wijze kunnen oplossen. Deze talen onderscheiden:

1. het geval waarin P2 niet bestaat,
2. het geval waarin P1 niet bestaat.

Oorspronkelijk kent FORTH slechts het geval 1 , met omkering van de zin van de voorwaarde.

Dit is de constructie:

BEGIN P1 V END

hetgeen als volgt moet worden gelezen:

Herhaal het geheel van procedures, symbolisch P1 genoemd , totdat de (voor)waarde op de stack achtergelaten TRUE (ongelijk aan nul) is en ga vervolgens na END verder. In PASCAL notatie

REPEAT P1 UNTIL V

Merk op dat P1 tenminste één keer wordt uitgevoerd.
Wil men echter de test uitvoeren vóórdat er iets wordt uitgevoerd, dan is er de volgende mogelijkheid

LOOP: (P1) V WHILE P2 REPEAT

hetgeen wordt gelezen als:

Zolang V waar is, herhaal het geheel van procedures, gesymboliseerd door P2 .

In PASCAL notatie:

WHILE V DO P2

In FORTH wijzigt de aanwezigheid van P1 vóór WHILE niets aan de lusopbouw. In andere talen, zoals ALGOL en PASCAL, is dit echter niet mogelijk.

Knuth stelt wel een dergelijke constructie als die in FORTH voor in deel 3 van zijn handboek "The Art of Computer Programming" :

LOOP: P1 WHILE V : P2 REPEAT

Evenals in FORTH kunnen P1 en P2 leeg zijn.
Let op het scheidingsteken : dat hier nodig is tussen V en P2 , terwijl door het gebruik van de Reverse Polish Notatie in FORTH er geen dubbelzinnigheid in de betreffende uitdrukking kan optreden.

Voorbeelden :

- (wacht lus) de procedure READY geeft FALSE (∅) zolang een instrument nog niet klaar is, vervolgens TRUE (1) als het dat wél is. Stel dat je wilt wachten tot het instrument klaar staat.

```
: WAIT BEGIN READY END ;
```

- een instrument telt events. Stel dat je de meting wilt beëindigen als er tenminste 1000 events zijn geweest.

```
: EXPERIMENT ∅ BEGIN ADDITION +
                     DUP
                     999 >
                END ;
```

- De grootste gemene deler (greatest common divisor GCD) van twee getallen wordt verkregen door
  - permutatie van de twee getallen (indien nodig), zodat het eerste het grootste is.
  - toepassing van de relatie

    GCD (a,b) = GCD (b,a-b)

    totdat a-b nul wordt.

  De twee getallen worden op de stack verondersteld

```
      : GCD BEGIN 2DUP  > IF SWAP THEN
                  OVER -
                  DUP Ø=
            END DROP ;
```

- Stel we willen een geheel getal omzetten in Romeinse cijfers:

```
BASE @ DECIMAL
: ROMAN BEGIN DUP 999  > IF ."M"  1ØØØ - WHILE
              DUP 499  > IF ."D"   5ØØ - THEN
        BEGIN DUP  99  > IF ."C"   1ØØ - WHILE
              DUP  49  > IF ."L"    5Ø - THEN
        BEGIN DUP   9  > IF ."X"    1Ø - WHILE
              DUP   4  > IF ."V"     5 - THEN
        BEGIN DUP   Ø  > IF ."1"     1 - WHILE
        DROP ;

BASE !
```


- Voorbeeld van gebruik van ROMAN :

```
: TABLE BASE @ DECIMAL CR CR ."TABLE OF POWERS OF 2 "
   CR CR 1Ø FLD !   1 BEGIN DUP DUP S. 2 BLANKS ROMAN CR
   2 * DUP 4ØØØ1  > END CR CR DROP  BASE !  ;

OK

TABLE

       TABLE OF POWERS OF 2

    1  I
    2  II
    4  IIII
    8  VIII
   16  XVI
   32  XXXII
   64  LXIIII
  128  CXXVIII
  256  CCLVI
  512  DXII
 1Ø24  MXXIIII
 2Ø48  MMXXXXVIII
 4Ø96  MMMMLXXXXVI
 8192  MMMMMMMMCLXXXXII
16384  MMMMMMMMMMMMMMMMCCCLXXXIIII

OK
```

- Reeks van Ulam

Zij een reeks getallen gegeven door de relatie

$$N_{i+1} = \begin{cases} Ni/2 & \text{indien } Ni \text{ is even} \\ 3Ni+1 & \text{indien } Ni \text{ is oneven} \end{cases}$$

De veronderstelling is: de reeks Ni convergeert altijd naar 1 na zekere tijd.

Stel dat we de reeks Ni willen afdrukken tot Ni = 1

```
1Ø CYCLE ICR  6 FLD !

: ULAM  1 ' ICR !
        BEGIN          WHILE
              DUP 1  > ~~IF~~ DUP
                          2 MOD IF    3 * 1 +
                                ELSE  2 /
                                THEN
                          DUP S.
                          ICR Ø= IF CR THEN
        ~~WHILE~~ REPEAT
        CR CR  DROP  ;

: RULAM Ø DO CR CR ."REEKS VAN ULAM VANAF " I S. CR
           I ULAM
          LOOP  CR CR CR ;

5 ULAM
    2Ø    1Ø    4     2    1

OK
33  ULAM
     122   51   174    76    37   136   57  216
     1Ø7  326   153   5Ø2   241   744  362  171
     554  266   133   422   211   634  316  147
     466  233   722   351  1274   536  257 1Ø16
     4Ø7 1426   613  2242  1121  3364 1572  675
    247Ø 1234   516   247   766   373 1362  571
    2154 1Ø66   433  1522   651  2374 1176  477
    1676  737  2636  1317  4156  2Ø67 6246 3123
   11372 4575 1617Ø  7Ø74  3436  1617 5256 2527
   1ØØØ6 4ØØ3 14Ø12  6ØØ5 22Ø2Ø 11Ø1Ø 44Ø4 22Ø2
    11Ø1 33Ø4  1542   661  2424  1212  5Ø5 172Ø
     75Ø  364   172    75   27Ø   134   56   27
     1Ø6   43   152    65   24Ø   12Ø   5Ø   24
      12    5    2Ø    1Ø     4     2    1

OK
```

```
12 RULAM

REEKS VAN ULAM VANAF      Ø

REEKS VAN ULAM VANAF      1

REEKS VAN ULAM VANAF      2
     1

REEKS VAN ULAM VANAF      3
    12     5    2Ø    1Ø     4     2     1

REEKS VAN ULAM VANAF      4
     2     1

REEKS VAN ULAM VANAF      5
    2Ø    1Ø     4     2     1

REEKS VAN ULAM VANAF      6
     3    12     5    2Ø    1Ø     4     2     1

REEKS VAN ULAM VANAF      7
    26    13    42    21    64    32    15    5Ø
    24    12     5    2Ø    1Ø     4     2     1

REEKS VAN ULAM VANAF     1Ø
     4     2     1

REEKS VAN ULAM VANAF     11
    34    16     7    26    13    42    21    64
    32    15    5Ø    24    12     5    2Ø    1Ø
     4     2     1

OK
```

4.6 Opmerkingen over het gebruik van herhalings- en vergelijkingsoperatoren

Deze operatoren maken deel uit van een klasse van operatoren die zich onderscheidt doordat deze operatoren moeten worden uitgevoerd bij de compilatie èn uitvoering van de procedure die hen bevat. (Het gaat dus duidelijk om twee verschillende aspecten van de operator).

Laten wij eerst kijken wat er tijdens uitvoering gebeurt. De variabele (of het register IC) bevat het adres van de volgende uit-te-voeren procedure (net zoals het P register van een computer het adres van de volgende instructie bevat).

De operatoren zoals DO , BEGIN en THEN veranderen niets aan IC en hebben bij de uitvoering geen equivalent. Zij dienen slechts voor de compilatie.

De andere (IF, END etc.) bevatten een sprong, al dan niet voorwaardelijk, naar voren, of naar achteren. Zij zijn op de volgende wijze geprogrammeerd:

blok 22

Het woord volgend op de operator (IF etc.) wordt aan IC toegevoegd of vervangt soms de inhoud van IC als de sprong plaats moet hebben. Zo niet, dan wordt eenvoudigweg steeds 1 bij IC opgeteld.

Bij de compilatie laat de eerste operator op de stack het adres achter van het volgende woord in het woordenboek. De tweede operator rekent de sprong uit en plaatst de verkregen waarde op de geschikte plek om bij de uitvoering te worden gebruikt.

Enkele versies van FORTH voeren tijdens de compilatie een eenvoudige test uit op deze operatoren: blok 1002

Aan iedere tweelingsoperator (of bij iedere drielingsoperator) wordt een willekeurig doch verschillend getal toegevoegd.

Zodra een linker operator (DO , BEGIN , etc.) wordt ontmoet, wordt het erbij behorende getal op een extra stack geplaatst. Wanneer een rechter- of sluithaakje wordt ontmoet (LOOP , END , etc.) wordt het getal op de top van deze extra stack aangepast waarna wordt gekeken of deze overeenkomt met de operator. Zo niet, dan is er een kruising tussen paren operatoren opgetreden, of een haakjespaar is niet gesloten of niet begonnen, hetgeen de drie meest voorkomende fatale fouten zijn.

Deze "SECURITY" versies van FORTH gaan ook na of genoemde operatoren uitsluitend binnen in een procedure worden gebruikt en niet in een interactieve modus. Ook wordt gecontroleerd of er niets op de stack wordt achtergelaten tijdens de compilatie.

Een voorbeeld van deze SECURITY zoals hij is gerealiseerd voor de HP 21MX wordt in appendix G behandeld. Met uitzondering van assembler routines en adres behandeling is deze beveiliging ook op identieke wijze geprogrammeerd voor een UNIVAC 1108 en enkele andere machines.

blok 26

4.7 Voorwaardelijke compilatie

Noot :

- Deze paragraaf behoort enerzijds thuis bij de voorwaardelijke sprongen en anderzijds bij hoofdstuk 5 als mogelijkheid in de meta-assembler, aangezien deze constructie toepasbaar is zowel op hoog niveau als op assembler niveau. Dit is één van de redenen waarom het moeilijk is om FORTH met klassieke systemen te vergelijken.

Tot nu toe hebben wij enkel kennis gemaakt met voorwaardelijke sprongen die binnen in een procedure-lichaam toepasbaar zijn.

Soms kan zich echter het geval voordoen dat er meerdere versies bestaan van een programma en dat je nu eens de ene en dan weer de andere versie wilt compileren. De reden hiervan is dat je om veiligheidsredenen in geval van veranderingen slechts één versie wilt bewaren, behalve in enkele gedeeltes, die best afwijkend kunnen zijn. Zoiets kan in het bijzonder optreden bij programma's die op twee - enigszins verschillende - installaties moeten werken (bijv. andere configuratie).

Hiervoor staan enkele voorwaardelijke compilatie-operatoren ter beschikking die de top van de stack testen op het moment dat de compilatie plaats vindt. Deze zijn:

```
IFTRUE
OTHERWISE
IFEND
```

Zij worden gebruikt op dezelfde wijze als IF ELSE THEN binnen in een procedure.

Het volgende voorbeeld illustreert dit. We schrijven een programmaatje om een paper tape te ponsen op een snelle ponser (PUNCH) of, indien er geen snell ponser is, op een teletype (TTY) . We gaan na of een snelle pons aanwezig is door op te vragen of er een entry PUNCH is

```
?DEF PUNCH IFTRUE
     PUNCH OTHERWISE TTY
     IFEND OTA,
```

blok 20

?DEF laat het adres van PUNCH achter als PUNCH reeds gedefinieerd
was
anders Ø

Geassembleerd wordt tenslotte ôf PUNCH OTA,
ôf TTY OTA, .

Een ander voorbeeld is het volgende. Het massageheugen (mass-storage) kan zich bevinden op een schijf (disc) of op magnetische band. Stel eens dat de blokken voor schijfgebruik vanaf blok 400 staan en voor bandgebruik vanaf blok 405. Nu zou je kunnen schrijven:

```
?DEF  MAG-TAPE  IFTRUE  405 LOAD
      OTHERWISE         ?DEF DISC
                        IFTRUE  400 LOAD
                        OTHERWISE  ." NO MASS STORAGE "
                        IFEND
      IFEND
```

5.1 De assembler

De FORTH assembler biedt behalve alle klassieke mogelijkheden zoals die door de fabrikant worden geleverd ook de mogelijkheden van een krachtige macro-assembler. De notatie is echter weinig gebruikelijk.

Evenals alles in FORTH werkt ook de assembler in Reverse Polish Notation, waarbij het adres, de index, indirect bit etc. aan de instructiecode voorafgaan.

De algemene werkwijze is als volgt:

- construeer eerst het volledige adres, met index, indirect bit etc. indien nodig, op de stack;
- de instructie gaat na of het adres in een geschikt gebied ligt, voegt de instructie-code aan het adres toe en plaatst het geheel in de eerstvolgende vrije locatie in het woordenboek.

  (In sommige gevallen treden page jumps op bij het bespelen van het geheugen; de assembler dient dan een link te leggen om van de ene "pagina" naar de andere te kunnen komen via base-page of via een speciale LINK table.)

Daar de hier geschetste werkwijze verschilt van die van de hogere taal, in FORTH, wordt de instructie van de assembler gevolgd door een komma (plaatsing van een woord in het woordenboek).

Het volgende, zeer eenvoudige, voorbeeld illustreert dit:

Stel dat je de inhouden van de variabelen V1 en V2 wilt optellen en het resultaat wilt plaatsen in V3 . In code geschreven ziet dat er zo uit:

```
V1   LDA,
V2   ADA,
V3   STA,
```

zet het adres van de variabele op de stack (↗ V3)

(↖ STA,) stelt de instructiecode samen en zet hem in het woordenboek

Merk nu op dat:

- alle mogelijkheden van FORTH ter beschikking staan voor de adresberekening, [in compilatietijd]
- er geen labels zijn (evenmin als in FORTH trouwens),
- de instructiecode niet beperkt is tot het plaatsen van één codeopdracht in het woordenboek. Men spreekt hier van een macro-assembler, die het mogelijk maakt om steeds herhaalde reeksen van instructiecode of speciale voorwaardelijke sprongen te laten genereren door de aanroep van een normale FORTH instructie.

  Aangezien dergelijke procedures weer kunnen worden opgenomen in andere procedures, is het duidelijk dat de mogelijkheden vrijwel onbeperkt zijn.

Vergelijk bv. de volgende code-opbouw in normale assembler en in FORTH :

LDA *-4	wordt	HERE 4 -	LDA,
LDA 4+V		V 4 +	LDA,

De kop van een assembler procedure wordt in de bibliotheek gemaakt door

CODE gevolgd door de naam van de procedure.

Merk hierbij op dat CODE nu STATE op Ø laat staan (EXECUTION). De pointer van de geassocieerde code wijst onmiddellijk naar de eerste instructie die op de kop volgt. De pointer kan worden gewijzigd, bv. om te wijzen naar het inwendige van een andere assembler procedure.

Als een assembler procedure locale parameters nodig heeft, dan kunnen deze vóór de kop worden geplaatst, of tussen de kop en de code, met de speciale constructie ,CODE .

Voorbeeld :

- Stel men wil de top van de stack steeds met 4Ø vermenigvuldigen. Dit kan op de volgende wijze gebeuren:

HERE 4Ø ,	zet 4Ø in het woordenboek en laat zijn adres op de stack achter
CODE 4Ø* S) LDA, ~~4Ø*~~ (~~MPY~~) ...	vermenigvuldigt met het woord waarvan het adres tijdens deze assemblage op de top van de stack staat.

Op analoge wijze wordt ,CODE gebruikt:

n ,CODE < naam > V1 , ... Vn , < instructies >

n ,CODE genereert een zodanige entry dat de pointer naar de geassocieerde code n plaatsen verder wijst dan bij CODE , om zodoende plaats te maken voor de n parameters V1 ... Vn . De operator < , > zorgt ervoor dat deze in het woordenboek worden opgenomen. De adressen van de parameters zijn dan bekend zodra het adres van bv. de eerste bekend is. Hiervoor gebruikt men de operator < ' > < naam > . Dit kan trouwens bij elke type entry. ↳ zet op de stek.
eventueel iets erbij optellen.

Het vorige voorbeeld kan nu ook zo worden geschreven:

```
1 ,CODE *4Ø 4Ø ,
        S) LDA, ' *4Ø MPY, ...
```

Hoe de tweede ?

5.2 Gebruik van de stack in assembler

Daar de meeste FORTH operaties plaats hebben via de stack moet men over middelen beschikken om in assembler code definities te maken die de stack snel toegankelijk maken.

Dit gebeurt met de volgende operatoren:

SP variabele bevat een pointer naar de top van de stack

SP1 idem voor het woord onder de top.

Hiermee zijn opgebouwd:

S) indirect naar de top van de stack

S1) idem naar het woord onder de top.

Voorbeeld :

- Stel dat we de twee woorden op de top van de stack willen optellen:

  S) LDA, S1) ADA,

- Stel dat we n woorden uit de stack willen weghalen, met n op de top

  S) LDA, INA, SP ADA, SP STA,

  [en n zelf ook extra weg.]

Let op :

- Als SP niet in een geheugenplaats maar in een indexregister is geplaatst, kan het gebruik iets veranderen, maar S) zowel als S1) zullen in het algemeen toch op de hier geschetste wijze worden gebruikt.

5.3 Hoe een assemblerprocedure te beëindigen

Aan het eind van een assemblerprocedure moet men:

- de stack netjes achterlaten
- zo snel mogelijk terugkeren naar de aanroepende procedure.

De meeste "gewone" manieren van terugkeren zijn hieronder vermeld. Elke assemblerprocedure moet met één uit deze lijst eindigen. (Een < * > voor de naam geeft aan dat deze niet op elke installatie aanwezig is).

	NEXT,	terugkeer zonder de stack te veranderen
	PUT,	} vervangt de top van de stack door de inhoud van register A
*	PUT,	
*	BPUT,	idem register B
	PUSH,	} plaatst de inhoud van A op de top van de stack
*	APUSH,	
*	BPUSH,	idem voor register B
	POP,	verwijdert één woord van de top van de stack
	POP.,	verwijdert twee woorden van de top van de stack
	BINARY,	} POP, gevolgd door PUT, (APUT,)
*	ABINARY,	
*	BBINARY	idem met BPUT,
*	BINARY.,	} POP., gevolgd door PUT, (APUT,)
*	ABINARY.,	
*	BBINARY.,	idem met BPUT,
	.NEXT,	voert onmiddellijk de procedure uit waarvan het adres in het B register is.

blok 486.
365
13

ABORT,

(ERROR,)

5.4 Lussen en voorwaardelijke sprongen in assembler

pas op

FORTH assembler beschikt over dezelfde lus- en voorwaardelijke operatoren als op het hogere niveau, qua notatie verschillend door de toegevoegde komma, qua implementatie gebruik makend van expliciete al dan niet voorwaardelijke sprong opdrachten.(Op sommige installaties kan de toegevoegde komma ontbreken. In die gevallen moet uit de context worden opgemaakt of men met FORTH dan wel FORTH assembler te maken heeft.)

Twee- of drievoudige constructies:

```
    BEGIN, ... END,
... IF, ... ELSE, ... THEN,
    BEGIN, ... WHILE, ... REPEAT,
```

END, IF, en WHILE, moeten worden voorafgegaan door een voorwaardelijke sprong-instructie.

Voorbeeld :

- Stel dat men bij een variabele Q steeds 5 wil optellen (waarbij het getal 5 in de variabele FIVE wordt bewaard) totdat Q $\geq \emptyset$ is geworden.

```
Q LDA,
BEGIN, FIVE ADA, A+, END, ...
```

Het gebruik van BEGIN, ... END, is gemakkelijk in te zien.
Hun definities zijn:

```
: BEGIN, HERE ;   zet op de stack het adres van de
                  eerstvolgende vrije plaats in het
                  woordenboek,

: END,   JMP, ;   compileer een sprong-opdracht naar
                  dat adres.
```

Doordat het (sprong-)adres op de stack wordt geplaatst, kan men een willekeurig aantal BEGIN, ... END, paren in elkaar definiëren (totdat de operationele stack het opgeeft!).

De operatoren IF, THEN, ELSE, LOOP:, WHILE, REPEAT, werken op analoge wijze.

In enkele gevallen hoeven de paren niet volledig te worden ingesloten, hetgeen een kruising van return adressen oplevert. Rectificatie geschiedt dan met SWAP .

5.5 De macro-assembler

Een macro-assembler vereenvoudigt de taak van de programmeur door het mogelijk te maken dat één symbool een hele reeks van instructies aanroept en de daarbij behorende instructiecodes genereert.

De twee volgende eenvoudige voorbeelden illustreren dit. Daar het merendeel van de minicomputers geen instructie heeft om rechtstreeks een waarde bij een geheugenlocatie op te tellen, maakt men gebruik van een register. (Stel dat A de waarde bevat)

```
ADA  MEM
STA  MEM
```

Wij gaan nu een procedure ADM, construeren die automatisch deze twee instructies genereert, terwijl de programmeur slechts een "gewone" instructie-opdracht gebruikt:

```
: ADM, DUP ADA, STA, ;
```

hetgeen wordt gebruikt als MEM ADM, . in andere definities.

Nu een tweede voorbeeld: Vaak moet men de pointers SP en SP1 tegelijk incrementeren. Hiervoor bestaat een macro-instructie ISP, :

```
: ISP, SP1 LDB, SP STB, INB, SP1 STB, ;
```

Midden in een assembler procedure genereert ISP, de drie machinecode instructies die SP en SP1 beide met 1 ophogen.

Opmerkingen :

- De macro-instructies zullen worden gebruikt in de procedures van het type CODE . Zij dienen bij voorkeur te worden geplaatst in de ASSEMBLER vocabulaire met de opdracht

  ASSEMBLER DEFINITIONS

  voorafgaand aan de definitie van de macro-instructie.
- Het aanroepen van een macro-instructie leidt tot de generatie van machinecode in het woordenboek. De code zelf wordt niet op dit moment uitgevoerd.

Het volgende voorbeeld licht dit toe:

Stel dat we aan IC het getal 1 willen toevoegen om bv. over één FORTH instructie heen te springen

```
CODE  SKIP  ONE  LDA,          1 # LDA,
            IC   ADM,
            NEXT,
```

De code in het woordenboek zal nu zijn:

```
ONE  LDA,
IC   ADA,
IS   STA,
NEXT,
```

Het gebruik van SKIP in een procedure van het type : leidt tot een sprong over de volgende procedure. Het zou natuurlijk ook mogelijk zijn geweest SKIP direct helemaal in machinecode te schrijven.

De voordelen van het toepassen van macro-instructies zijn:

- compacte en duidelijke programmacode zonder wijziging van de echt uitgevoerde code.
- de overdraagbaarheid naar andere machines is groter. Vaak is de macro-instructie eenvoudig om te zetten van de ene machine op de andere.

Het voorgaande voorbeeld is op zichzelf te eenvoudig om het gebruik van van macro-instructies te rechtvaardigen.

Laten we een tweede voorbeeld geven:

De code voor */ is :

```
CODE  */  S1)  DLD,
          B    MPY,
          S)   DIV,
               ISP,
          BINARY,
```

In de gegenereerde code wordt ISP, vervangen door

```
SP1  LDB,  SP  STB,  INB,  SP1  STB, .
```

Stel eens dat het FORTH pakket enigszins wordt veranderd en dat SP in een indexregister zou worden bevat. De programmacode voor */ hoeft nu *niet* te worden veranderd. De macro-instructie ISP, moet wel worden veranderd en de uiteindelijke machinecode zal dan ook anders zijn.

Bekijk tenslotte de volgende procedure:

```
CODE ISP SP1 LDB, SP STB, INB, SP1 STB, NEXT,
```

of nog eenvoudiger geschreven (de gegenereerde codes zijn identiek)

```
CODE ISP ISP, NEXT ,
```

Nu is ISP geen macroconstructie, maar een operator die bij aanroep SP en SP1 met 1 ophoogt, dus direct wordt uitgevoerd. ISP is derhalve equivalent aan DROP . Zij kan niet worden gebruikt om in een CODE procedure machine-instructies te genereren.

6.0 De verschillende typen variabelen

FORTH heeft een relatief grote keus in typen variabelen, Sommige daarvan, zoals de integers, de floating points, de vectoren en de matrices, zijn traditioneel. Andere zijn dat minder, zoals het type polynoom. Bovenal maakt FORTH het eenvoudig mogelijk om nieuwe typen variabelen te creëren voor bepaalde problemen, alsook de bewerkingen op die variabelen.

In FORTH is de stack dé plaats waar de operatoren hun informatie halen of deponeren. Wij maken nu onderscheid tussen variabelen die

- een gegeven (een constante bv.) op de stack plaatsen, waarbij de variabelen zelf niet veranderd moeten worden, of
- op de stack het adres van een variabele achterlaten, waarbij de waarde van de variabele wel veranderd zal worden.

Dit onderscheid wordt in FORTRAN niet expliciet gemaakt, is echter van fundamentele betekenis in FORTH .

6.1 De standaard variabelen

a) Eenvoudige variabelen

< waarde > CONSTANT < naam >

< waarde > ~~INTEGER~~ VARIABLE < naam >

Deze twee constructies maken een entry met de naam van de variabele en zetten de waarde die op de stack is op het moment van compilatie als parameter in de bibliotheek.

Bij gebruik van deze variabelen plaatst de toegevoegde (of geassocieerde) code de *waarde* van de parameter op de stack van een CONSTANT , of het adres voor een INTEGER .

Het adres van een constante kan worden verkregen door

' < naam van de constante >

De inhoud van een INTEGER kan worden verkregen door

< naam van de INTEGER > @

Voorbeeld :

```
-    Ø INTEGER  SPEED
     6Ø CONSTANT MAXSPEED

   : SNELHEIDSMETER  MEET  SPEED  !  ;
   : OVERTREDING  SPEED  a  MAXSPEED  >  ;
```

SNELHEIDSMETER meet de snelheid (resultaat op de stack) en zet de snelheid in SPEED , OVERTREDING vergelijkt SPEED met MAXSPEED .

b) Dubbele woord variabelen

< dubbele waarde > DCONSTANT < naam >

< dubbele waarde > DOUBLE < naam >

Deze constructies zijn analoog aan die voor CONSTANT en INTEGER maar met *twee* woorden als parameters.

c) Vectoren

< maximum index > ARRAY < naam >

< maximum index, dubbele waarden > 2ARRAY < naam >

0 -> n

Deze twee constructies zijn van type "variabele". Zij reserveren n+1 of 2(n+1) woorden in het parameterveld in het woordenboek.

Gebruik hiervan:

< waarde van de index > naam
laat op de stack achter het adres van het i-de onderdeel.

d) Vector constant

VECTOR < naam > waarde 1 , waarde 2 , etc.

De dimensie wordt hier niet expliciet meegegeven. Men zet de waarden (constanten) achtereenvolgens in het woordenboek na de kop.

Bij het gebruik ervan moet de waarde van de index aanwezig zijn in de variabele ( ) (van type INTEGER) .

Voorbeeld: ∅ VARIABLE ( ) is reeds gedef.

VECTOR CENTRE ∅ , 1∅ , 6 ,
VECTOR APEX 2∅ , 2 , 4 ,
etc.
∅ ( ) SET X 1 ( ) SET Y 2 ( ) SET Z

Om de afstand in X tussen APEX en CENTRE te krijgen
X CENTRE APEX -

Opmerking:

- Voor het gebruik van SET zie 6.3
- De variabele ( ) is op elk moment gemeenschappelijk aan alle variabelen van type VECTOR .

e) "Polynomen" constanten.

< aantal waarden > POLY < naam > < waarde 1 > , etc.

< aantal waarden, tweevoudige bestanddelen > 2POLY < tweevoudige waarde > , < 2de tweevoudige waarde >, etc.

Toepassing:

Elke aanroep van < naam > zet op de stack de waarde van het volgende bestanddeel. Nadat de laatste van de rij is bereikt, komt de eerste weer.

Voorbeeld:

4 POLY ANGLES Ø , 3Ø , 6Ø , 9Ø ,

Het aanroepen van ANGLES zet achtereenvolgens op de stack
Ø 3Ø 6Ø 9Ø Ø 3Ø etc.

f) Matrices

< maximale index rijen > < maximale index kolommen >

MATRIX < naam >

evenzo 2MATRIX < naam >

lijken op ARRAY en 2ARRAY maar hebben twee indices nodig in-
plaats van één.

6.2 Constructie van nieuwe types

Bij het maken van nieuwe types moeten twee problemen worden opgelost:

1. Het construeren van de kop en het parameterveld van de nieuwe variabele in het woordenboek.
2. Het maken van de geassocieerde code.

a) Constructie van de kop en het parameterveld

De kop kan worden gemaakt met behulp van één van de reeks gedefinieerde variabelen. Meestal wordt hiervoor CONSTANT' gebruikt, maar soms ook wel een ander type om bv. een groter parameterveld te reserveren.

Zo is bv. DCONSTANT gedefinieerd als

```
: DCONSTANT  CONSTANT  ,  ;CODE  ...
```

maakt de kop met het woord op de top van de stack als 1-ste parameter (CONSTANT)

voegt het tweede woord toe als 2-de parameter (,)

Gewoonlijk wordt de geassocieerde code in assembler geschreven. Er is ook een andere manier, nl. om de toegevoegde code in de hogere taal te schrijven.
In dit geval moet de definitie eindigen met < ;:' > in plaats van ;CODE . < ;:' > bevat al de constructie van een kop met CONSTANT . Als meerdere woorden moeten worden gereserveerd, dan moet dat gebeuren vóór de kop en niet in het parameterveld.

Voorbeeld :

- De definitie van MATRIX is

: MATRIX

OVER 1+ ,	zet in het woordenboek het aantal lijnen +1 ,
1+ SWAP 1+ *	berekent het totaal aantal te reserveren woorden
HERE	pakt de actuele plaats in het woordenboek en zet dit op de stack

SWAP DP +!	voegt aan DP het aantal benodigde woorden voor de matrix toe
;:	eindigt de constructie van de kop. De geassocieerde pointer wijst naar de daaropvolgende hogere code. De eerste parameter is het adres van het eerste woord van de matrix. Zodra een matrix gebruikt gaat worden, wordt déze waarde op de stack geplaatst, net zoals bij een normale constante. CONSTANT maakt als het ware deel uit van de toegevoegde code.

b) <u>Constructie van de geassocieerde code</u>

Blok 46

De geassocieerde code volgt onmiddellijk op ;CODE als de programmacode in assembler of ;: als hij in hogere code wordt geschreven.

In assembler bevat het B-register, bij overgang naar de geassocieerde code, het adres van het woord voorafgaande aan de eerste parameter.

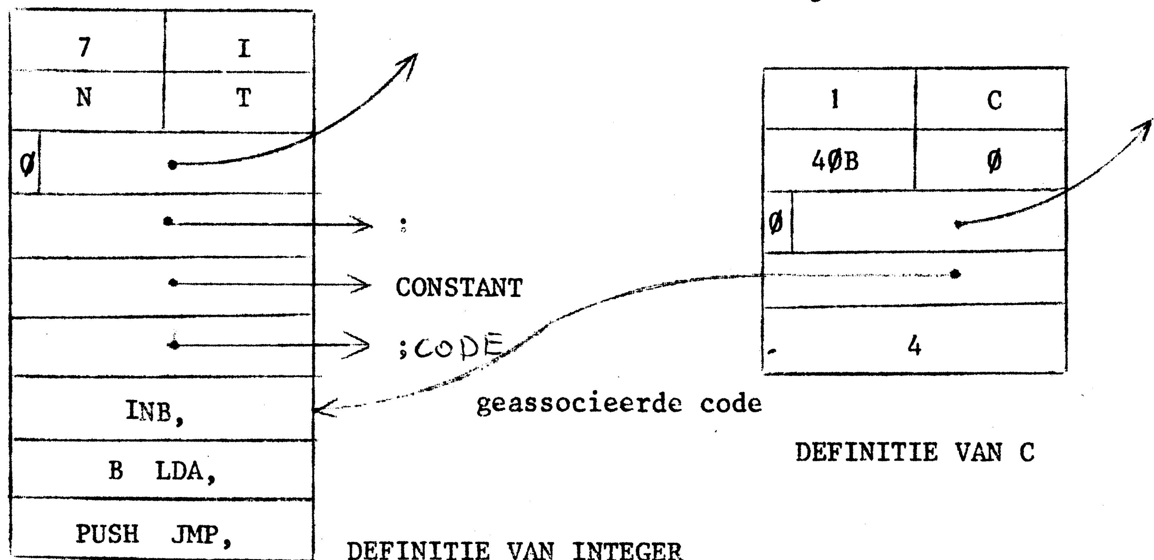
<u>Voorbeeld</u> :

INTEGER is gedefinieerd door:

Blok 391

```
: INTEGER  CONSTANT  ;CODE
   INB,     adres van de eerste parameter
   B LDA,   wordt verplaatst
   PUSH,    naar de top van de stack.
```

Wat gebeurt er bij 4 INTEGER C ?
In het geheugen staat dan:



Blok 372

N.B. ;CODE wijzigt automatisch de aan C geassocieerde pointer. zodanig dat deze verwijst naar de code onmiddellijk volgend op ;CODE .

Een ander voorbeeld:
Stel dat we een nieuw type variabele willen creëren genaamd CYCLE . Dit moet zich gedragen als een constante maar wordt met 1 opgehoogd na iedere aanroep. Vervolgens wordt de bereikte waarde modulo n genomen, waarbij n eigen is aan een bepaalde variabele. We willen een entry maken met twee parameters, de actuele waarde van de constante en de modulo waarde n , evenals de code die voor de uitvoering zorg moet dragen.

HERE -1 ,	-1 wordt in het woordenboek geplaatst met zijn adres op de stack
1501 : CYCLE Ø CONSTANT , ;CODE	
INB, B) LDA,	plaats de actuele waarde in A
B)* ISZ,	incrementeer de actuele waarde
INB,	we willen nu n hebben. (het adres van n is in B)
B) CPA, RSS,	vergelijk met n
PUSH,	de waarde n is nog niet bereikt, zet de actuele waarde op de stack

ADB,	decrementeer B met 1 (zie hierboven HERE -1 ,)
CLA, B) STA,	zet de waarde terug op nul
B) ISZ,	incrementeer deze
PUSH,	zet nul op de stack.

Voorbeeld van het gebruik van CYCLE :

Stel dat wij binnen een LOOP: ... WHILE ... REPEAT lus 1Ø waarden op een regel willen afdrukken.
Gebruik nu: 1Ø CYCLE ICR

ICR heeft gewoonlijk de beginwaarde Ø .
We zetten deze waarde op 1 met de opdracht

1 ' ICR ! vervolgens LOOP: ... WHILE ...

S. ICR Ø= IF CR THEN

... REPEAT

Geassocieerde code met behulp van de hogere taal

Bij geassocieerde code in hogere taal wijst de *waarde* van de constante naar eventuele parameters.

Begonnen op 84.

Laten wij nu het voorbeeld van MATRIX afmaken:

;:	(het begin van de toegevoegde code)
SWAP	verwisselt de adressen van het aantal rijen en kolommen
OVER 1- @	zet het aantal rijen op de stack
* + +	berekent het adres van het element dat op de stack staat
;	hier wordt de toegevoegde code beeindigd.

6.3 De werkwoorden

De werkwoorden vormen een heel bijzondere klasse van procedures waarvoor geen enkel equivalent in enig andere taal te vinden is. Bij een type variabele horen niet alleen waarden maar ook een specifieke handeling gepleegd op deze waarden.

Bijvoorbeeld: wij willen bij iedere aanroep van een werkwoord van een bepaald type, aan een variabele een gegeven waarde toekennen, zoals bv.:

OCTAL dient om 8_{10} te plaatsen in BASE
DECIMAL dient om 10_{10} te plaatsen in BASE

Wij zouden ook kunnen definiëren:

```
: OCTAL 8 BASE ! ;
```

Aangezien deze actie regelmatig voorkomt, heeft men deze geoptimaliseerd. Stel dat er een constructie is

< waarde > < adres > SET < naam >

bijvoorbeeld : 1Ø BASE SET DECIMAL

De procedure SET maakt nu een entry met DECIMAL als naam. Iedere aanroep van DECIMAL zet 10_{10} in BASE . De aan DECIMAL toegevoegde code moet aanwezig zijn in SET .
SET wordt gedefinieerd als:

```
: SET SWAP CONSTANT , ;CODE
```

Er wordt een kop gemaakt met twee parameters; de waarde zelf en het adres van de variabele.

```
INB,          haalt de waarde op (eerste parameter)
B ) LDA,
INB,          haal het adres (tweede parameter)
B ) LDB,      naar B ,
B ) STA,      plaats de waarde op het adres van de variabele en
NEXT,         keer terug.
```

De nieuwe definitie van OCTAL is veel sneller dan de oude. Ook neemt zij minder plaats in in het woordenboek.

6.4 Reeksen van karakters (character strings)

Reeksen van karakters zijn nog niet gestandaardiseerd in FORTH . De hier volgende beschrijving is voorlopig en komt overeen met de implementatie op ESO en de Observatoire de Genève. Zij verschilt aanzienlijk van de structuur zoals die wordt voorgesteld door Philip Hill van de Sterrewacht van St. Andrews.

`." < EEN TEKST > "` — drukt deze tekst af bij compilatie, als de constructie buiten een procedure lichaam is, of bij uitvoering / executie } als zij in een procedure is.

Noot :

- In de hier volgende constructies kan voor de hier gebruikte delimiter < % > elk ander symbool worden gebruikt (behalve de spatie die als algemeen woordscheider wordt gebruikt).

`STRING % TEKST % NAAM` — maakt een entry NAAM wijzende naar TEKST . Deze kan worden afgedrukt met de opdracht NAAM TYPE

`N STRINGARRAY % TEKST N-1 % ... % TEKST Ø % NAAM` — maakt een tabel met pointers naar de teksten. Deze kunnen worden afgedrukt met het commando

I NAAM TYPE (Ø <= I < N)

`ABORT-STRING % TEKST % NAAM` — maakt een kop die naar de tekst verwijst. Bij het uitvoeren van NAAM wordt er een system-abort gegeven en deze TEKST afgedrukt. De system-abort stopt de in uitvoering zijnde procedure.

`N ABORT-STRING-ARRAY % TEKST N-1 % ... ... % TEKST Ø % NAAM` — maakt een entry van pointers naar de teksten. Het uitvoeren van I NAAM drukt de TEKST I af en stopt de executie.
Ø <= I < N

7.0 De normale communicatie met FORTH (Input-Output)

De communicatiemogelijkheden van een systeem zijn vaak moeizaam te definiëren. Er is in feite een voortdurend conflict tussen

- gebruiksgemak - eenvoud van bediening;
- algemeenheid, overdraagbaarheid;
- uitvoeringssnelheid.

FORTH ontsnapt niet aan deze regel.

Het basis FORTH definieert slechts twee typen van input-output (I/O) :

- met de gebruikersconsole
  (teletype, scherm, etc.)
- met een massageheugen
  (schijf, magnetische band, etc.).

Andere I/O typen worden overgelaten aan de keuze van de programmeur. Implementaties zullen sterk afhangen van de beschikbare I/O structuren, van de snelheidsbeperkingen of van de overdraagbaarheid van het systeem. Er zullen zeker verschillende oplossingen zijn voor een onafhankelijk systeem (stand-alone system) of voor een FORTH werkend in een door de fabrikant geleverd systeem (zoals RTE), die in het algemeen de I/O structuur vastlegt op systeem-niveau.

Er is een voorstel in de maak om een protocol te definiëren, dat de implementatie voor de programmeur doorzichtig maakt.

Die oplossing zou altijd moeten worden toegepast als de snelheidseisen dit toelaten.

De "stand-alone" versie van FORTH die de programmeur toegang geeft tot alle mogelijkheden van de machine, zonder beperkingen, en in het bijzonder tot de I/O, biedt zulke unieke mogelijkheden voor het afregelen van instrumenten en hun "drivers" tot aan de limiet van logica en materiaal, dat men het zich niet kan veroorloven hier geen kennis van te willen nemen.

7.1 Commando's voor het afdrukken

a) Veranderlijken voor het definiëren van (eventuele) formats

FLD bevat de lengte van het veld,

DPL bevat het aantal decimalen na de punt.

Deze twee variabelen zijn van type INTEGER ; zij kunnen onafhankelijk van elkaar worden geherdefinieerd, of met de opdracht

< lengte van veld > < aantal decimale plaatsen > W.D

N.B. Na het inlezen van getallen bevatten deze twee variabelen de waarden overeenkomend met het ingelezen getal.

b) Typ opdrachten

CR geeft een terug wagen ",CR," en nieuwe regel, "LF" . Indien het systeem werkt met een buffergeheugen (zoals bij RTE) dan wordt de gehele regel pas afgedrukt na de opdracht CR .

37 . drukt de waarde op de top van de stack af, gevolgd door één spatie (onafhankelijk van F en D).

S. evenzo, maar links aanvullend (right justified) in een veld met lengte FLD

O. als . maar in OCTAL , onafhankelijk van de waarde in BASE .

? drukt de inhoud af van het adres op de top van de stack (gelijk aan @.)

D. drukt de waarde af van het dubbele getal op de top van de stack volgens FLD en DPL .

F. drukt de waarde af van het "floating" getal op de top van de stack volgens FLD en DPL .

SPACE	geeft een spatie
< m > SPACES	geeft m spaties
< m > CRS	geeft m regels
39 ." text "	drukt de text af tussen ." en "
37 < m > WCH	schrijft het ASCII character waarvan de waarde op de stack staat.

< character adres > < aantal characters > TYPE

drukt het aantal characters af vanaf het gegeven adres .

< adres van een kop > COUNT

zet de parameters goed voor TYPE om de naam van een
392 entry af te drukken.
(Dit wordt o.a. gebruikt bij kettingen van characters).

< blok nummer > LIST

40 drukt de 16 lijnen van een blok af en nummert ze.
(Zie voor blokstructuur het massageheugen in hoofdstuk 8.0).

< begin blok > < eind blok > BLIST

1503 **drukt alle blokken af tussen de twee aangegeven limieten.**

< begin blok > < eind blok > CLIST

1502 drukt de eerste regel af van alle blokken tussen de twee aangegeven limieten.

WHERE drukt de naam af van de kop van de laatstgemaakte
1502 entry in de bibliotheek, eventueel gevolgd door de naam van de vocabulaire waarin dit is gebeurd.

c) Verandering van uitvoertoestel

PRINTER leidt de afdruk naar de snelle drukker als die bestaat.

TERMINAL brengt de afdruk weer terug naar de "terminal" van
37 de gebruiker.

VLIST list bieb.

100 HERE DUMP dump geheugen.

8.0 Virtueel en massageheugen

Reeds vanaf zijn ontstaan heeft FORTH gebruik gemaakt van een massageheugen op schijfband of op een magnetische cassette.
Tegenwoordig gebruikt FORTH ook wel een operating system dat hem herbergt. Deze versie lijkt slechts uit de verte op het oorspronkelijke FORTH, maar biedt in het algemeen samen met het operating system een sterkere beveiliging.
In het oorspronkelijke FORTH wordt het massageheugen verdeeld in 512 blokken van 512 16-bit woorden. De blokken zijn genummerd van 1 tot 512. Soms worden er nog meer genummerd, bijv. met schijfgeheugens.

In het centrale geheugen (central memory, vroeger ook CORE memory genoemd) zijn tenminste twee gebieden gereserveerd voor de inhoud van deze blokken, die zowel gegevens als programma's kunnen bevatten.

Het oproepen van deze blokken om bijv. een copie te maken in het massageheugen, de keuze van welk blok te gebruiken bij een nieuwe aanroep, is geheel automatisch. Dit komt overeen met een virtueel geheugen dat voor programmeren op systeemniveau beschikbaar is en niet slechts op het niveau van de programmeur.

Vijf variabelen of procedures maken het mogelijk deze blokken te behandelen:

447 BLK is een variabele van type INTEGER , die het bloknummer bevat van een blok voor gebruik met de "editor".

BLOCK is een procedure welke het bloknummer op de top van de stack vervangt door het adres van het eerste woord van dit blok in het geheugen. Als het blok niet in central memory is, wordt het eerst gecopieerd (opgehaald) vanuit het massageheugen.

446 UPDATE is een procedure die aangeeft dat de blokken zijn veranderd en opnieuw moeten worden gecopieerd naar het massageheugen vóórdat zij in het central memory worden overschreven door andere blokken.

447 FLUSH is een procedure die een transfer maakt tussen blokken in het centrale geheugen en het massageheugen - als de UPDATE flag van de blokken is gezet.

460

LOAD haalt dat blok naar central memory wiens nummer op de stack staat en begint de text van dit blok te interpreteren. Deze kan op zijn beurt weer andere blokken "laden" en compileren, met als enige restrictie de stacklengte.

Enkele eenvoudige voorbeelden

4 BLK ! we willen blok 4 gebruiken

7 BLOCK @. druk af de inhoud van het eerste woord van blok 7

: BLOCK-DUMP BLOCK DUMP-OFFSET ! Ø 1ØØØ DUMP CR ;

5 BLOCK-DUMP list de inhoud van blok 5 in een speciaal format.

Stel dat we de blokken in kleinere databestanden willen onderverdelen met behulp van

de konstanten RECORD-LENGTH
#-RECORDS/BLOCK

Stel dat we een procedure ADDRESS willen hebben dat het nummer van een RECORD op de stack vervangt door het adres van het eerste woord van de RECORD - na het blok te hebben opgehaald uit het massageheugen, als dit nodig is. Dan kunnen we dit schrijven als

: ADDRESS #-RECORDS/BLOCK /MOD 511 MIN BLOCK SWAP RECORD-LENGTH * + ;

V.1 De berekening van e met een groot aantal decimalen in een gegeven grondtal (base)

De berekening van e of π met een zeer groot aantal significante cijfers (enkele duizenden) heeft altijd aantrekkingskracht gehad op mathematen en op programmeurs, zelfs al heeft dit geen enkel fundamenteel belang meer.

De hier voorgestelde methode gebruikt geen meervoudige precisie operatoren. De voor e verkregen waarde kan echter wel worden gebruikt om meervoudige precisie-operatoren te toetsen wanneer zij gebruikt worden in een meer bekend algorithme.

Algorithme van Sale

Stel een reeks-ontwikkeling van e^x gegeven door :

(1) $$e^x = \sum_{i=0} \frac{x^i}{i!}$$

(2) $$= 1 + x \left[1 + \frac{x}{2} \left[1 + \frac{x}{3} (1 + \ldots) \right] \right]$$

Door $x = 1$ te stellen krijgen we e

$$e = 2 + \tfrac{1}{2} \left[1 + \frac{1}{3} (1 + \ldots) \right]$$

Men moet echter de sommatie beperken tot een eindig aantal (m) termen met behulp van de vergelijking

(3) $$m \text{ min } : m! > b^{(n+1)}$$

waar n het aantal decimalen is in grondtal (base) b .

Deze relatie kan niet worden berekend voor grote m . In dat geval wordt m! benaderd door de formule van Stirling :

(4) $$m . \left[\ell n(m) - 1 \right] + 0.5 \, \ell n \, (2.\pi.m) > (n + 1) \, . \, \ell n \, (b)$$

Voorbeeld: we krijgen - m = 73 voor b = 10 , n = 100
- m = 1000 voor b = 10 , n = 2566 .

Laten we nu het aantal decimalen van e berekenen met het algorithme van Sale. In dit voorbeeld gebruiken wij b = 10 .
Vermenigvuldig teller en noemer van (2) met b :

(5) $e = 2 + \frac{1}{10} \left| \frac{1}{2} \left(10 + \frac{1}{3} \left(10 + \frac{1}{4} (10 + \ldots) \right) \right) \right|$

Haal nu binnen de haken het gehele en het fractionele gedeelte uit elkaar.

(6) $e = 2 + \frac{1}{10} \left| 7 + \frac{1}{2} \left(\emptyset + \frac{1}{3} \left(10 + \frac{1}{4} (0 + \ldots) \right) \right) \right|$

Het gehele getal stelt de nieuwe decimaal voor.
Merk op dat het gehele getal wordt verkregen door eerst het binnenste haakjespaar uit te werken. Om de volgende decimaal te krijgen moeten we weer teller en noemer vermenigvuldigen met het grondtal b .

Uitwerken van het programma

Wij definiëren eerst enkele constanten.

EBASE grondtal waarin wordt gerekend en afgedrukt

ND aantal gewenste decimalen

M aantal termen van de reeks, berekend volgens (4)

COEF een ARRAY met dimensie ≥ M dat de tellers van (6) bevat naar gelang de berekening verder gaat.

We willen het afdrukken netjes maken om het lezen te vergemakkelijken.
De hoofdprocedure neemt de volgende vorm aan:

```
: ECALC EINIT
        ND 1+ 1 DO DIGIT S.
                   CPRINT
                LOOP ;
```

EINIT initialiseert de tabel COEF en de verschillende variabelen nodig voor het afdrukken van de getallen.

DIGIT berekent een nieuwe decimaal.

CPRINT regelt een nieuwe pagina en lay-out van de pagina.

DIGIT berekent de nieuwe decimaal volgens relatie (b), telt op de stack de gehele deeltallen op en vervangt iedere coëfficiënt op de stack door de rest na de deling. We beginnen met ∅ op de stack.

```
: DIGIT Ø 2 M DO I COEF @
                 EBASE * +
                 I /MOD
                 SWAP I COEF !
            -1 +LOOP ;
```

Opmerking :

- De lusindex neemt af van M tot en met 2 , met increments van -1 ,
- Met de operator /MOD wordt een tijdelijke variabele vermeden, evenals enige "delikate" operaties op de stack.

Tenslotte maken we de volgende lay-out voor het afdrukken:

10 groepen van 5 cijfers per lijn
10 groepen van 5 lijnen per pagina

De variabelen van het type CYCLE lenen zich bij uitstek voor dit soort besturing. Wij definiëren vier variabelen:

```
 5 CYCLE CBL      5 CYCLE CLI
10 CYCLE CRC     10 CYCLE CPG
```

De initialisatie kan als volgt geschieden:

```
: EINIT  M  1+  2  DO  1  I  COEF  !  LOOP
         5  F  !  2  S.  ."  .  "
         1  F  !
         1  '  CBL  !    1  '  CRC  !
         1  '  CLI  !    1  '  CPG  !  ;
```

De lay-out kan dan zo gaan:

```
: CPRINT  CBL  Ø=  IF  ."  "
                  CRC  Ø=  IF  CR
                            CLI  Ø=  IF  CR
                                      CPG  Ø=  IF  PAGE  THEN
                                  THEN  "  ...  "
                            THEN
                  THEN  ;
```

Het programma is hiermee beëindigd. We kunnen tenslotte aan het begin van EINIT de berekening van M toevoegen volgens relatie (4) .

Opmerkingen :

- Als MAX het grootste gehele getal is dat op de stack mogelijk is, moet er voldaan zijn aan
  EBASE * M = MAX .
- BASE moet in overeenstemming zijn met EBASE .
- In plaats van één enkel cijfer per keer te berekenen kun je er meerdere tegelijk berekenen als volgt.
  Je neemt voor EBASE een macht van het grondtal waarin wordt afgedrukt. (Bijv. EBASE = 1000 voor het berekenen van drie getallen tegelijk.) Er moet dan wel een andere initialisatie en lay-out komen. De berekeningstijd neemt af naarmate er meerdere cijfers gelijktijdig worden berekend. In grondtal 2 kunnen we gemakkelijk 5 cijfers gelijktijdig berekenen. De beperking van het aantal dat maximaal gelijktijdig kan worden berekend volgt uit de eerste opmerking daarover.

```
DECIMAL 10 CONSTANT EBASE  1000 CONSTANT M M ARRAY COEF
     2566 CONSTANT ND

5 CYCLE CBL  5 CYCLE CLI  10 CYCLE CRC  10 CYCLE CPG

: EINIT M 1 + 2 DO 1 I COEF ! LOOP 5 F ! 2 S. " ." 1 F !
       1 ' CBL !  1 ' CRC !  1 ' CLI !  1 ' CPG !  ;

: DIGIT 0  2 M DO I COEF @ EBASE * +
                  I /MOD SWAP I COEF !
               -1 +LOOP  ;

: CPRINT CBL 0= IF "  "
                   CRC 0= IF CR
                             CLI 0= IF CR
                                       CPG 0= IF PAGE THEN
                                    THEN "        "
                          THEN
                THEN ;

: ECALC CR CR EINIT
       ND 1 + 1 DO DIGIT S. CPRINT LOOP
       CR CR ;
```

PROGRAMME POUR LE CALCUL DE E AVEC BEAUCOUP DE DECIMALES

EBASE	INDIQUE LA BASE SERVANT POUR L'IMPRESSION
M	INDIQUE LE NOMBRE DE TERMES DU DEVELOPPEMENT
ND	INDIQUE LE NOMBRE DE CHIFFRES A IMPRIMER

L'IMPRESSION SE FAIT A RAISON DE 10 GROUPES DE 5 LIGNES PAR PAGE
10 GROUPES DE 5 CHIFFRES PAR LIGNE

L'ALGORITHME UTILISE SE TROUVE DECRIT DANS L'ARTICLE :
THE CALCULATION OF E TO MANY SIGNIFICANT DIGITS
DE A. H. J. SALE (1968), THE COMPUTER JOURNAL, V:11 P229-230

```
OK
600 ' ND ! ECALC


   2.71828 18284 59045 23536 02874 71352 66249 77572 47093 69995
     95749 66967 62772 40766 30353 54759 45713 82178 52516 64274
     27466 39193 20030 59921 81741 35966 29043 57290 03342 95260
     59563 07381 32328 62794 34907 63233 82988 07531 95251 01901
     15738 34187 93070 21540 89149 93488 41675 09244 76146 06680

     82264 80016 84774 11853 74234 54424 37107 53907 77449 92069
     55170 27618 38606 26133 13845 83000 75204 49338 26560 29760
     67371 13200 70932 87091 27443 74704 72306 96977 20931 01416
     92836 81902 55151 08657 46377 21112 52389 78442 50569 53696
     77078 54499 69967 94686 44549 05987 93163 68892 30098 79312

     77361 78215 42499 92295 76351 48220 82698 95193 66803 31825
     28869 39849 64651 05820 93923 98294 88793 32036 25094 43117

OK
```

V.2 Onderzoek naar permutaties van een vector in hun lexicografische ordening

Het probleem wordt als volgt gedefinieerd:

Gegeven is een vector Z: van dimensie NVAL

We willen 1) als gegeven is dat er inderdaad nog één bestaat, de permutatie van Z: hebben, die onmiddellijk in lexicografische zin (alfabetische waarde) op de voorgaande volgt.

2) deze procedure gebruiken om alle permutaties van de letters F O R T H te maken.

Een onderzoek van het eerste probleem:

Om de volgende permutatie in lexicografische ordening te krijgen moeten we proberen zoveel mogelijk de waarde "links" te behouden en slechts termen "rechts" te veranderen (permuteren).

Het eerste element om te wijzigen is het i-de, zō dat i de grootste index is met $i < nval$

en Z : (i) < Z : (i+1)

Het bestaan van i is verzekerd door het "bestaan van nog een permutatie".

We zoeken de volgende permutatie en vervangen nu Z : (i) door het kleinste element dat nog rechts van Z : (i) is, nl.
j : $i < j \leqslant nval$

en Z : (j) is het kleinste element waarvoor geldt

Z : (j) > Z : (i) → Z: is steeds een woord.

We moeten nog de overige elementen in het rechtergedeelte ordenen.

Zij ID de procedure die de waarde i vaststelt
JD de procedure die de waarde j vaststelt
i j Z:SWAP de procedure die Z : (i) en Z : (j) verwisselt
i nval Z:QUEUE de procedure die de Z : (k) ordent
$i < k \leqslant nval$

De volgende permutatie van Z: wordt verkregen door

```
: Z:PERMUT   ID    JD
                   OVER   Z:SWAP
             1+    NVAL   Z:QUEUE  ;
```

Noot :

- ID en JD laten respectievelijk i en j op de stack achter, OVER Z:SWAP permuteert Z:(i) en Z:(j) maar laat i op de stack achter, die wordt geïncrementeerd en daarna opnieuw wordt opgepakt met NVAL door Z:QUEUE .

Nu moeten we de vier gebruikte procedures opschrijven.

```
: ID  NVAL  1-  LOOP:   DUP   Z:  @
                        OVER  1+  Z:  @
                        <  ∅=  WHILE  1-
                REPEAT ;

: JD  NVAL      LOOP:   Z:  @
                        SWAP  Z:  @
                        >  ∅=  WHILE  1-
                REPEAT ;
```

Noot :

- Merk op dat in beide procedures het gebruik van LOOP: ... WHILE ... REPEAT moeilijk te omzeilen is. Vaak zal men uit de lus springen, zelfs vòòrdat eenmaal de lus doorlopen is, in het bijzonder als i = nval-1 of j = nval .

  ∅= wordt hier gebruikt om de zin van de voorwaarde om te keren (< ∅= is equivalent aan ⩾) .

```
: Z:SWAP    Z:    SWAP    Z:
            OVER  @       OVER  @
            SWAP  ROT     !  ROT  !  ;
```

en tenslotte

```
: Z:QUEUE   LOOP:  2DUP
                   <  WHILE   2DUP  Z:SWAP
                              1-  SWAP
                              1+  SWAP
            REPEAT
            DROP  DROP  ;
```

Opmerkingen :

- De vorige noot geldt ook voor Z:QUEUE
- De operatie die de ordening uitvoert komt in dit geval overeen met een inversie van de "rechter" termen.

Oefeningen :

- Gebruik NVAL = 4 , Z: = 1, 2, 3, 4
  en voer met de hand de bewerking Z:PERMUT uit.
  Houd intussen alle tussenresultaten op de stack bij.
- Wat gebeurt in Z:QUEUE als het aantal termen dat geordend moet worden even dan wel oneven is?
- Modificeer Z:PERMUT (of nog beter de procedures die door Z:PERMUT worden aangeroepen) zodanig dat in plaats van de onmiddellijk volgende permutatie, de direct voorafgaande wordt verkregen.
- De procedure ID (ook JD werkt op dergelijke wijze) komt overeen met

  i = nval - 1

  zolang Z:(i) ⩾ Z:(i+1) decrementeer i ;

  Wat gebeurt er als men het ⩾ teken vervangt door >
  En wat gebeurt er bij JD ?
- Wat gebeurt er als twee elementen uit Z: gelijk zijn?
- Zowel bij ID als bij JD gaat men uit van een beginwaarde (i = nval - 1 , j = nval) welke men decrementeert zolang niet aan de voorwaarde is voldaan. Loop je geen kans om negatieve indices te krijgen?

Laten we nu het tweede gedeelte van ons probleem bekijken, en wel het gebruik van Z:PERMUT .

Het is mogelijk vrijwel direct een procedure PERM-FORTH te schrijven als we ons de voorwaarde herinneren waaraan Z:PERMUT moet voldoen, d.w.z. dat de volgende permutatie bestaat. Zij Z:>ORDRE de procedure die waar is (TRUE) wanneer Z: volledig in afnemende ordening is (en dus niet verder in lexicografische zin geordend kan worden, d.w.z. de laatste letter in alfabetische volgorde vooraan, de vóórlaatste daarvoor etc.)

```
:  PERM-FORTH   Z:INIT-FORTH  Z:PRINT
               LOOP:  Z:>ORDRE
                      Ø=    WHILE   Z:PERMUT
                                    Z:PRINT
               REPEAT  ;
```

Deze procedure volgt de algemene regels van dergelijke programma's:

- na een initialisatie gaat men zolang er werk te doen is door, om tenslotte de resultaten af te drukken.

Om andere ensembles van waarden te permuteren, hoeven we alleen Z:INIT-FORTH te veranderen.

Laten we nu snel de laatste procedures bekijken

```
:  Z:INIT-FORTH     4     NVAL  !
                    1Ø6   Ø  Z:  !   11Ø  1  Z:  !
                    117   2  Z:  !   122  3  Z:  !
                    124   4  Z:  !
                    CR    CR
                    ."  PERMUTATIONS DES LETTRES FORTH"
                    CR  CR  ;

:  Z:>ORDRE         1     NVAL  Ø  DO  I  Z:
                                       I  1+  Z:
                                       >  AND
                                   LOOP   ;

:  Z:PRINT          6     BLANKS
                    NVAL  1+  Ø  DO
                                     I  Z:  WCH  BLANK
                                 LOOP
                    CR  ;
```

<u>Opmerkingen</u> :

- In FORTH loopt de index van Ø tot n', waarna de waarde wordt doorgegeven aan NVAL.
- De ondergrens van de index van Z: verschijnt voor de eerste keer expliciet in Z:INIT-FORTH , en daarna ook in Z:>ORDRE .

- In al de voorgaande routines hadden we een limiet kunnen vaststellen van 1 of een willekeurige andere waarde (zelfs negatief!)
- Z:>ORDRE speelt een fumdamentele rol
  a) waakt tegen verboden gebruik van Z:PERMUT .
  b) beëindigt de taak wanneer alle permutaties zijn afgedrukt.
- In Z:INIT-FORTH worden de letters FORTH gegeven in hun alfabetische volgorde (d.w.z. FHORT).

Oefeningen :

- Probeer door wijziging van deze nieuwe procedures om de permutaties van FORTH in afnemende rangorde te laten gebeuren.
- Bij de hierboven gegeven initialisatie krijgt men 120 (=5!) permutaties. Hoeveel permutaties zijn er als je uitgaat van FORTH (in plaats van van FHORT)?
- Wijzig Z:PERMUT zo dat je *altijd* een nieuwe permutatie krijgt. We voegen dan eerst de (cyclische) conventie toe dat de eerste permutatie in monotoon groeiende ordening (de eerste dus in lexicografische ordening) onmiddellijk volgt op de laatste permutatie (de laatste dus in lexicografische zin).

Noot : Er ontbreekt nog een initialisatie van NVAL en van Z: Dit zou bijvoorbeeld kunnen met

```
10 CONSTANT NVAL NVAL ARRAY Z:
```

Historische opmerking :

Dit voorbeeld is in feite ontleend aan het werk van Dijkstra (cf. bibliografie). De algorithme zelf is al in 1812 gepubliceerd in Dresden door Fischer en Krause, in hun boek 'Lehrbuch der Kombinationslehre und der Arithmetik', en is vrijwel identiek aan de algorithme van Dijkstra.

Het algemene probleem van permutaties (en niet alleen in lexicografische zin) heeft tot een zeer groot aantal publicaties geleid, zowel in de mathematica als in de informatica. Het 2e nummer, volume 9, van de "ACM Computing Surveys" (juni 1977) is helemaal gewijd aan een artikel van R. Sedgewick over dit onderwerp. Behalve een gedetailleerde be-

schrijving van de belangrijkste van de bekende algoritmen, bevat het artikel ook een studie over de (soms spectaculaire) optimatisatie die hierin mogelijk is op het niveau van de machine code evenals een uitgebreide bibliografie.

PERM-FORTH

PERMUTATIONS DES LETTRES F O R T H

1	F H O R T	2	F H O T R	3	F H R O T
4	F H R T O	5	F H T O R	6	F H T R O
7	F O H R T	8	F O H T R	9	F O R H T
10	F O R T H	11	F O T H R	12	F O T R H
13	F R H O T	14	F R H T O	15	F R O H T
16	F R O T H	17	F R T H O	18	F R T O H
19	F T H O R	20	F T H R O	21	F T O H R
22	F T O R H	23	F T R H O	24	F T R O H
25	H F O R T	26	H F O T R	27	H F R O T
28	H F R T O	29	H F T O R	30	H F T R O
31	H O F R T	32	H O F T R	33	H O R F T
34	H O R T F	35	H O T F R	36	H O T R F
37	H R F O T	38	H R F T O	39	H R O F T
40	H R O T F	41	H R T F O	42	H R T O F
43	H T F O R	44	H T F R O	45	H T O F R
46	H T O R F	47	H T R F O	48	H T R O F
49	O F H R T	50	O F H T R	51	O F R H T
52	O F R T H	53	O F T H R	54	O F T R H
55	O H F R T	56	O H F T R	57	O H R F T
58	O H R T F	59	O H T F R	60	O H T R F
61	O R F H T	62	O R F T H	63	O R H F T
64	O R H T F	65	O R T F H	66	O R T H F
67	O T F H R	68	O T F R H	69	O T H F R
70	O T H R F	71	O T R F H	72	O T R H F
73	R F H O T	74	R F H T O	75	R F O H T
76	R F O T H	77	R F T H O	78	R F T O H
79	R H F O T	80	R H F T O	81	R H O F T
82	R H O T F	83	R H T F O	84	R H T O F
85	R O F H T	86	R O F T H	87	R O H F T
88	R O H T F	89	R O T F H	90	R O T H F
91	R T F H O	92	R T F O H	93	R T H F O
94	R T H O F	95	R T O F H	96	R T O H F
97	T F H O R	98	T F H R O	99	T F O H R
100	T F O R H	101	T F R H O	102	T F R O H
103	T H F O R	104	T H F R O	105	T H O F R
106	T H O R F	107	T H R F O	108	T H R O F
109	T O F H R	110	T O F R H	111	T O H F R
112	T O H R F	113	T O R F H	114	T O R H F
115	T R F H O	116	T R F O H	117	T R H F O
118	T R H O F	119	T R O F H	120	T R O H F

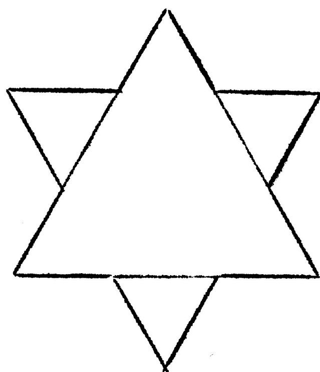
OK

V.3 Voorbeeld van een recursieve procedure

De kromme van Von Koch

Deze figuren werden op het einde van de vorige eeuw onderzocht door de mathematicus Von Koch ten behoeve van bepaalde integratieproblemen. Het zijn krommen die, in het limietgeval, oneindig lang zijn, maar die een volstrekt eindig en beperkt oppervlak omsluiten.

Beschouw een gelijkzijdige driehoek. Verdeel iedere zijde in drieën en construeer een nieuwe driehoek op het centrale segment. Vervolgens bewaren we slechts de "buiten"figuur.



We voeren dezelfde constructie uit op ieder segment van de nieuwe figuur, en zo vervolgens voort tot in het oneindige.

Merk op dat we na de n^{de} iteratie beschikken over

$$3 \,.\, 4^n \text{ segmenten ,}$$

en dat de lengte van ieder segment gelijk is aan

$$\frac{1}{3^n} \text{ van de zijde van de oorspronkelijke driehoek } (= 1)$$

Derhalve groeit de totale lengte van de kromme met

$$3 \,.\, \left(\frac{4}{3}\right)^n$$

De oppervlakte van ieder toegevoegd driehoekje is dan

$$\frac{\sqrt{3}}{4} \,.\, \left(\frac{1}{3^n}\right)^2$$

en er zijn hiervan $3 \,.\, 4^{n-1}$ die tezamen een oppervlakte leveren van

$$3 \,.\, 4^{n-2} \sqrt{3} \left(\frac{1}{3^n}\right)^2$$

Deze vormen derhalve een geometrische reeks met rede

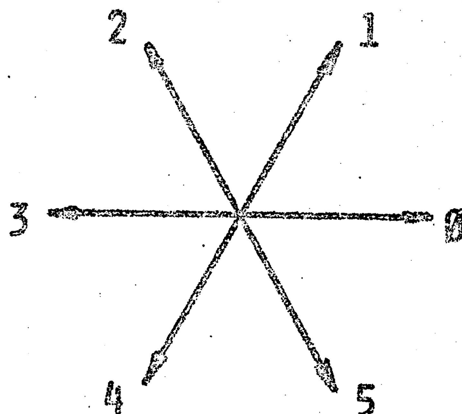
$\frac{4}{9}$

In de limiet zal de oppervlakte van de figuur dus $\frac{9}{5}$ zijn van die van het oorspronkelijk oppervlak.

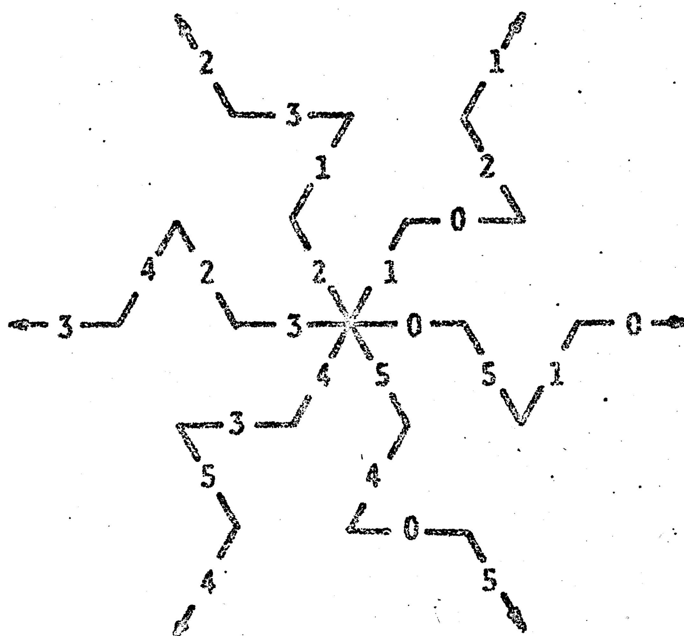
De oppervlakte zal trouwens altijd liggen binnen de begrenzing van de cirkel door de hoekpunten van de oorspronkelijke driehoek.

Wij willen nu een procedure maken waarmee we de kromme van Von Koch kunnen tekenen voor verschillende waarden van n .

Beschouw eerst de zes verplaatsingsrichtingen die we nummeren van ∅ tot 5 .



Stel bovendien dat we de kromme willen tekenen in de richting van tegen de wijzers van de klok in (↺) , laten we dan nagaan hoe ieder segment bij iedere iteratie wordt vervangen.



Ieder segment met richting i wordt vervangen door:

- een segment met richting i
- gevolgd door een segment met richting i-1 modulo 6
- gevolgd door een segment met richting i+1 modulo 6
- gevolgd door een segment met richting i

(we definiëren dat -1 modulo 6 = 5)

We kunnen nu een procedure PLVK samenstellen die een segment moet tekenen van een elementaire lengte-eenheid in de richting gegeven door de waarde op de stack (0 tot en met 5) en een procedure VKOCH die de recursieve opbouw vaststelt. Deze procedure vraagt twee parameters van de stack :

- op de top de index van recursiviteit
- daaronder de richting van het beschouwde segment.

Als de index van recursiviteit Ø is, dan laten we het segment tekenen.
Zo niet, dan roepen we VKOCH *recursief* aan met een vermindering van de recursiviteitsindex met 1 .

De beginwaarde van de recursiviteitsindex geeft de maximale diepte aan die tenslotte zal worden bereikt.

```
: VKOCH DUP Ø= IF
          DROP
          PLOT
       ELSE
          1-
          OVER OVER MYSELF
          OVER 1- 6 MOD OVER MYSELF
          OVER 1+ 6 MOD OVER MYSELF
          OVER OVER MYSELF
       THEN
   ;
```

Om de volledige kromme te tekenen, moeten we drie keer VKOCH aanroepen in de drie richtingen van de oorspronkelijke driehoek :
Stel dat de beginwaarde van de recursiviteitsindex als vrije parameter gelaten wordt voor de aanroep van VONKOCH

```
: VONKOCH  Ø  OVER  VKOCH
           2  OVER  VKOCH
           4  OVER  VKOCH
           DROP  ;
```

Het is ook mogelijk de driehoeken te construeren gericht naar de binnenkant van de figuur in plaats van naar de buitenkant.

```
: VONKOCHINT   1  OVER  VKOCH
               5  OVER  VKOCH
               3  OVER  VKOCH
               DROP  ;
```

Om een plotprocedure PLVK te construeren merken we op dat een hoek van 60^0 gemakkelijk gemaakt kan worden met incrementen (4,7) overeenkomend met $60^0 15'$. We maken twee tabellen DX en DY van ieder zes elementen voor de richtingen genummerd Ø tot en met 5 .

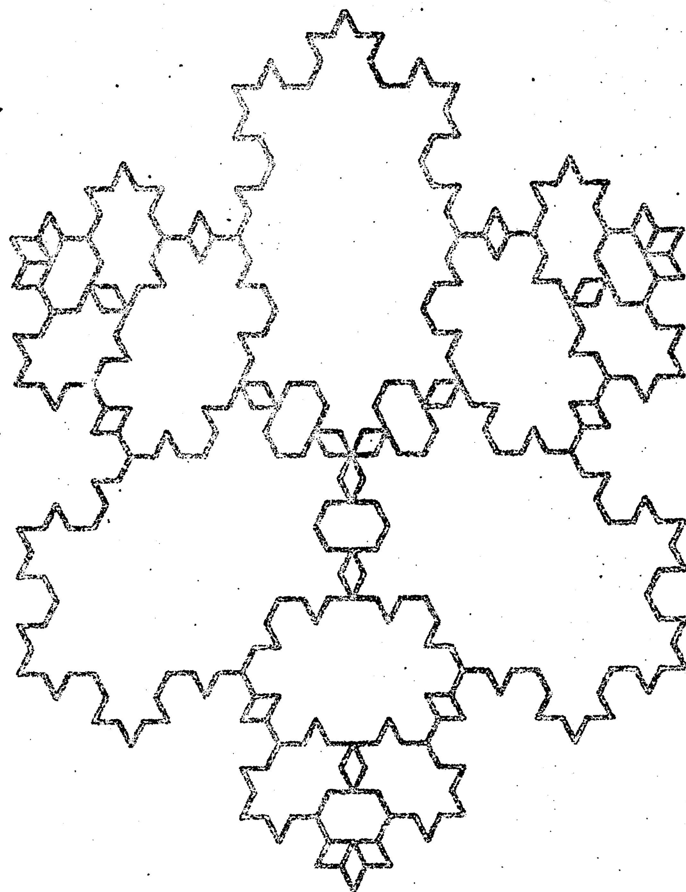
```
5 ARRAY DX -6 DP+!  1Ø,  4,  -4,  -1Ø,  -4,  4,
5 ARRAY DY -6 DP+!   Ø,  7,   7,    Ø,  -7, -7,
: PLVK  DUP  DX  @  SWAP  DY  @  PLT  ;
```

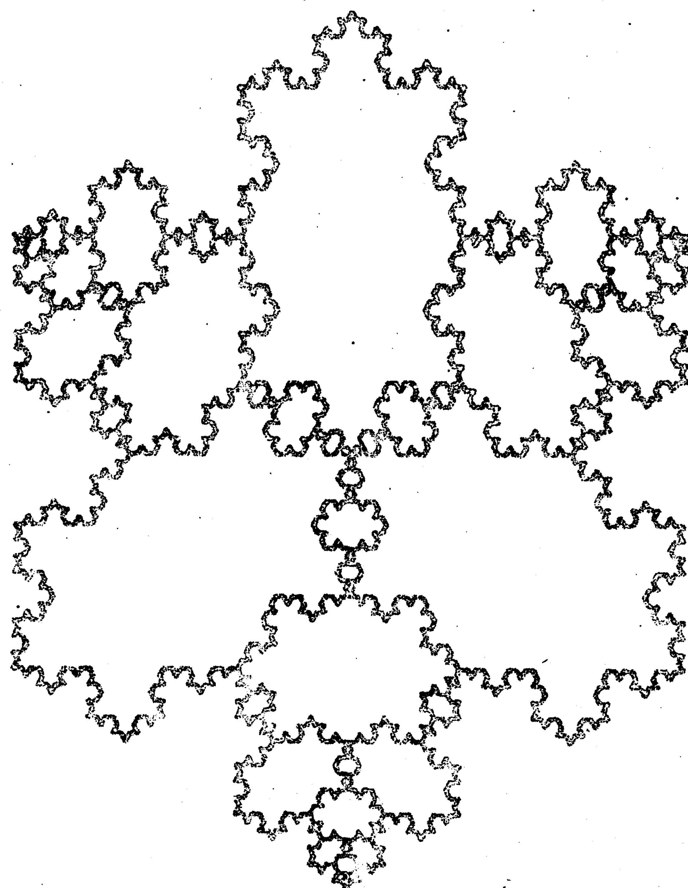
Mocht tenslotte op een machine -1 6 MOD niet tot antwoord 5 leiden, dan kunnen we zelf een procedure schrijven met het beoogde doel.

```
: 6MOD  6  MOD  DUP  Ø<  IF  DROP  5  THEN  ;
```

en deze 6MOD aanbrengen in VKOCH in de plaats van de oorspronkelijke 6MOD .

Figuren van Von Koch verkregen door de "externe" en "interne" krommen over elkaar te leggen voor recurrentie-diepten van 4 respectievelijk 5 .





V.6 Lijst van hoekpunten en ribben van een hyperkubus in een n-dimensionale ruimte

Stel dat we een foto-electrisch meetsysteem hebben met n filters en dat de metingen behept zijn met een onzekerheid. Zij $\frac{\varepsilon}{2}$ de maximale fout die gemaakt kan worden bij de een of andere kleur. Wij kunnen dan een hyperkubus definiëren voor een representatie van de maximale fout. Deze kubus is gecontreerd op de meetwaarde en heeft ribben ter lengte ε . Een analyse van de waarnemingen wordt in het algemeen gedaan met behulp van een projectie op een vlak van telkens twee lineaire, verschillende combinaties van de waargenomen kleuren. Wij willen nagaan hoe de zojuist gedefinieerde hyperkubus zich hierbij afbeeldt.

In feite zullen we slechts het eerste gedeelte van het probleem beschouwen door met aftellen en zonder herhalingen een nummer toe te kennen aan

- alle hoekpunten
- alle ribben

} van de hyperkubus

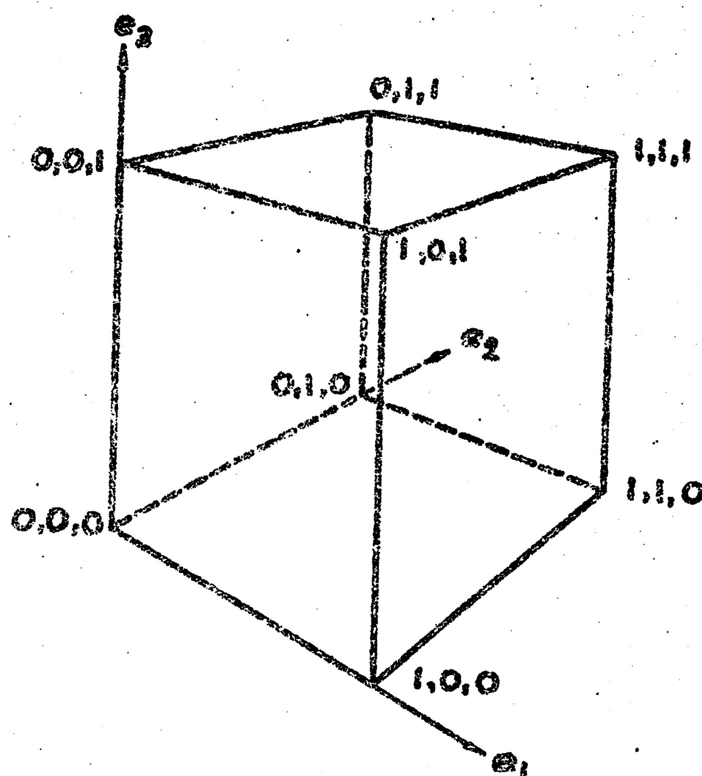
Merk allereerst op dat in een n-dimensionale ruimte een hyperkubus 2^n hoekpunten heeft, welke we kunnen nummeren van Ø tot 2^{n-1} . Dit stelt ons in staat de hoofdprocedure te schetsen:

```
: NDCUBE  2  N  ↑  Ø  DO
                         I  SOMMET  (hoekpunt)
                         I  ARETES  (ribben)
                   LOOP  ;
```

(De uiteindelijke versie verschilt slechts door de controle van het afdrukken en de lay-out van de pagina.)

In SOMMET berekenen we de coördinaten van het i^{de} hoekpunt en drukken deze af. Om het voorbeeld eenvoudig te houden leggen we de oorsprong van het coördinatensysteem op de hoek genummerd Ø en nemen we $\varepsilon = 1$ als lengte voor alle zijden.

De coördinaten van een hoekpunt zijn dan een mengsel van n *nullen enen.* Er zijn 2^n verschillende combinaties van Ø en 1 om de coördinaten van een hoekpunt aan te geven. Deze overeenkomst (equivalentie) suggereert een representatie in grondtal 2 voor de coördinaten van het i^{de} punt.



De coördinaten van de hoekpunten van een kubus (n = 3) .

We kunnen voor de hoekpunten SOMMET schrijven:

```
: SOMMET N Ø DO
              DUP 1 AND ,
              2/
          LOOP
       CR DROP ;
```

SOMMET drukt de coördinaten van het i^{de} punt af volgens de zojuist genoemde conventie.

We krijgen nu te maken met de procedure ARETES die alle ribben moet aanwijzen die uitgaan van het i^{de} hoekpunt en die nog niet zijn gebruikt. Een ribbe is per definitie evenwijdig aan één van de coördinaat-assen en loodrecht op alle andere assen, of, anders gezegd: alle coördinaten blijven onveranderd op één na, die van Ø in 1 verandert, of van 1 in Ø .

```
: ARETES  N  DO
               DUP
               I  TESTBIT
               Ø=  IF  DUP
                       I  SETBIT
                       SOMMET
                   THEN
           LOOP
        CR  DROP  ;
```

SOMMET wordt hier weer gebruikt om het tweede hoekpunt van de ribbe af te drukken.

De procedures TESTBIT en SETBIT worden in machine code (assembler) gedefinieerd. De actie die ze uitvoeren is de volgende:

I en M zijn waarden op de stack. $Ø \leqslant I \leqslant 15$

M I TESTBIT laat op de stack de waarde van het I^de^ bit van M achter.

M I SETBIT zet het I^de^ bit van M op 1 .

M I CLEARBIT zet het I^de^ bit van M op Ø .

Laten we eindigen met de uiteindelijke versie en met het resultaat verkregen door 3 in N te plaatsen, en NDCUBE aan te roepen.

```
6  CONSTANT  N              :  2↑N  1  N  Ø  DO  2  *  LOOP  ;
:  SOMMET  N Ø  DO
                DUP  1  AND  .
                2 /
            LOOP
        CR  DROP  ;
:  ARETES   N  Ø DO
                DUP
                I  TESTBIT
                Ø=  IF  DUP
                        I  SETBIT
                        "      VERS  "  SOMMET
                    THEN
            LOOP
        CR  DROP  ;
```

```
:  CTITRE   CR   CR  "  SOMMETS ET ARETES D'UN  "
            N          2 CASE " CARRE"
                 ELSE  3 CASE " CUBE"
                 ELSE           " HYPERCUBE A " N . " DIMENSIONS"
                 THEN  THEN
            CR   CR  ;

:  NDCUBE   CTITRE
            2+N Ø DO
                       I "  SOMMET   " SOMMET CR
                       I ARETES
                   LOOP
            CR CR ;

3  ' N  !    NDCUBE

 SOMMETS ET ARETES D'UN CUBE
 SOMMET   Ø Ø Ø

    VERS  1 Ø Ø
    VERS  Ø 1 Ø
    VERS  Ø Ø 1

 SOMMET   1 Ø Ø

    VERS  1 1 Ø
    VERS  1 Ø 1

 SOMMET   Ø 1 Ø

    VERS  1 1 Ø
    VERS  Ø 1 1

 SOMMET   1 1 Ø

    VERS  1 1 1

 SOMMET   Ø Ø 1

    VERS  1 Ø 1
    VERS  Ø 1 1

 SOMMET   1 Ø 1

    VERS  1 1 1

 SOMMET   Ø 1 1

    VERS  1 1 1

 SOMMET   1 1 1


OK
```

Y.1 Bibliografie

Bartholdi, P.

"Stepwise development and debuging using a small well structured interactive language for data acquisition and instrument control"

Mimi 76, 117-122, Zurich (1976)

Colin, A.J.T. et al.

"The translation and interpretation of STAB-12"

Software-Practice and Experience 5, 123-138 (1975)

Dahl, O.J., Dijkstra, E.W., Hoore, C.A.R.

"Structured Programming"

Academic Press N.Y. (1972)

Dewar, R.B.K.

"Indirect Threaded Code"

Comm. ACM 18, 330-331 (1975)

Dijkstra, E.W.

"A discipline of programming"

Prentice-Hall N.J. (1976)

Ewing, M.S.

"The Caltech FORTH Manual"

An Internal Report of the Owens Valley Radio Observatory (1976)

Iliffe, J.K.

"L'ordinateur à language de base"

Dunod, Paris (1970)

Jensen, K., Wirth, N.

"PASCAL User Manual and Report"

Springer Verlag, Lecture Notes in Computer Science 18, 2e Ed. (1975)

Knuth, D.E.

"The art of computer programming"

Addison-Wesley Vol. 1, 2, 3 (197x)

Moore, C.H. and Rather, E.D.

"The FORTH program for spectral line observing"

Proc. IEEE 61, 1346-1349 (1973)

Moore, C.H.

"FORTH : A new way to program a mini-computer"

Astron. Astrophys. Suppl. 15, 497-511 (1974)

Moore, C.H.
"The use of FORTH in process control"
International 77 Mini-Micro Computer Conference, Genève (1977)

Rather, E.D. and Moore, C.H.
"The FORTH approach to Operating Systems"
ACM 76 Proc. 233-240 (1976)

Rather, E.D. and Moore, C.H.
"FORTH high-level programming technique on microprocessors"
Electro/76 Professional Program, Boston (1976)

Rather, E.D. and Moore, C.H.
"An application oriented language"
Programmer's Guide, FORTH Inc. (1976)

Rather, E.D. and Moore, C.H.
"Mini-computer programming in FORTH"
Comm. ACM (in press 1977)

Sachs, J.M.
"STOIC, a Stack oriented Interactive Compiler"
Comm. ACM (in press 1977)

Stevens, W.R.
"A FORTH Primer"
Kitt Peak, Tucson (1977)

Wirth, N.
"The programming language PASCAL"
Acta Informatica 1, 35-63 (1971)

Wirth, N.
"Program development by stepwise refinement"
Comm. ACM, 14, No. 4, 221-227 (1971)

Wirth, N.
"Systematic programming : An introduction"
Prentice-Hall N.J. (1973)

Wirth, N.
"Algorithms + Date Structure = Programs"
Prentice-Hall N.J. (1976)

For real-time-monitor used in HP-FORTH systems see the description in

Brinch Hansen, P.
"Operating systems principles"
Prentice-Hall N.J. (1973)

For a discussion on real-time interactive-responding systems see

Martin, J.
"Design of man-computer dialogues"
Prentice-Hall N.J. (1973)

Z.1 Index

Opmerkingen :

- De entries in deze index zijn in lexicografische orde gedefinieerd door de ASCII-code.
- De entries in hoofdletters hebben betrekking op FORTH woorden.

(spatie)	90
!	11-14-21-80
." ... "	90
#	23-54
*	21
*/	21
+	21
+!	11-14
+LOOP	52-58-59
,	39-70-71
,*	21
,/	21
,CODE	49-72
-	21
-!	11-14
-LOOP	58
.	90
.NEXT	45-74
/	21
/MOD	21-96
Ø<	23-54
Ø=	23-54
ØSET	12-14
1+	12-14-21
1+!	11-14-21
1-	12-14-21
1-!	12-14-21
1IMP	36-39
1SET	12-14
2DUP	10-13
2OVER	11-13
2POP	74
2ROT	11-13
2SWAP	11-13
:	28-32-44-46
:ORPHAN	41-43
:ORX	41-43
;	28-44-47
;:	84-86
;CODE	49-83 e.v.
<	23-54
=	23-54
>	23-54
?	90
?DEF	39
@	11-14-21-80
ABINARY	74
ABORT	53
ABORT-STRING	88
ABORT-STRINGARRAY	88
ABS	21
ADOPT	39-41
ADOX	43
ALOG	24
analyse top-down	29 e.v.
AND	23

Vervolg

APUSH 74
APUT 74
ARRAY 80-82
ASSEMBLER 25-48-49
ATAN 24

BACKLOOP 58
BASIC 7
BBINARY 74
BEGIN 53-61-66
BEGIN, 75
bibliotheek 8-25-32
BINARY 74
bit, précédence 26-32
BLK 92-93
blok 91-92 e.v.
BLOCK 92-93
BNOT 23
bottom up coding 29 e.v.
BPUSH 74
BPUT 74

CASE 52-56-57
CHAIN 51
CODE 49-71
code geassocieerd 26-41-45-83-86
compilatiën 32-68
compilatiën, voorwaardelijke 68 e.v.
compilatiën, incrementeel 31
COMPILE 39
CONSTANT 35-80-83
CONTEXT 48-49
COS 24

Vervolg

COUNT 91
CR 90
CRS 90
CURRENT 48-49
CYCLE 85-86

D! 11-14-21
D* 21
D+ 21
D- 21
D. 90
D/ 21
D@ 11-14-21
databestand (massageheugen) 92 e.v.
DCONSTANT 80-83
DECIMAL 87
DEFINE 36-38
DEFINITIONS 49-51
dialect 51
dictionaire 8-25-32
DO 17-37-52-58-59-66
dubbel 19
DOUBLE 80
DP 9-39
DP! 39
DP+! 39
DP@ 39
DPL 90
DROP 10-13
DUP 10-13

e 94
ELSE 41-52-53-55-56-57
ELSE, 75
END 53-61
END, 75
entry 26 e.v., 41-48-70-72

Vervolg

ESAC 57
executie 32-34-35
EXITLOOP 53-58

F! 21
F* 21
F+ 21
F- 21
F. 90
F/ 21
FACULTEIT 36-59
FIX 22
FLD 90
FLOAT 22
floating point 19
FLUSH 92
functie, arithmetische 24
FORGET 40-51
format 90
FORWARD 36-38
fractie 19

gcd 63
geheel getal 19

HERE 39-71
HEX 87
hypercube 111

IC 35-45-47-66
IF 52-55-66
IF, 75
IFEND 68-69
IFTRUE 68-69
INTEGER 80-84-85
interface 31
interpreter 7

Vervolg

kop 26-71

label 53
LAST 32
LIST 91
LIT, 34
litterals 32
LOAD 93
LOOP 52-58-59

macro instructie 71-76
massageheugen 92 e.v.
MATRIX 82-83-86
MAX 21
MIN 21
MINUS 21
MOD 21
multiple precisie 94
MYSELF 36

naam 26-27-28-36-48-71-72-88
NDROP 10-13
NEXT 44-46-47-74
notatie, Poolse 19-20
NOVER 11-13

O. 90
OCTAL 87
operator, arithmetische 21 e.v.
operator, logische 23
OR 23
ORCX 41-43
ORPHAN 41-43

Vervolg

OTHERWISE 68-69
OVER 11-13

parameter 10-26-37
permutatie 99
PI 24
pointer 26
POLY 81-82
POP 74
PRINTER 91
procedure 28 e.v. -32-41
programmatie, hiërarchische 28-44
programmatie, gestructureerde 52
PUSH 74
PUT 74

recursiviteit 18 - 36 e.v. - 106
reeksen van characters 88
REPEAT 52-53-55-62-64
roman 63
ROT 11-13
RP 9-17-18
RP+! 17
RPØ 9-17
RP@ 17

S. 90
S) 73
S1) 73
S@ (11)-14
SDUMP 59-60
security 66
SET 87
simulatie 31

Vervolg

SIN 24
SP 9-10-13-73
SP! 12
SP+! 12
SPØ 9-10-13-73
SP@ 10-12
SPACE 90
SPACES 90
SQRT 24
stack, operationele 8-10 e.v.-73
stack, return 8-17 e.v.-37
STATE 32-71
stepwise refinement 28-31
STRING 88
STRINGARRAY 88
SWAP 11-13
synoniem 27-36-40

table de multiplication 59-60
TASK 40
TERMINAL 91
test 28-31
THEN 52-55-66
THEN, 75
top down design 29-31
TYPE 91

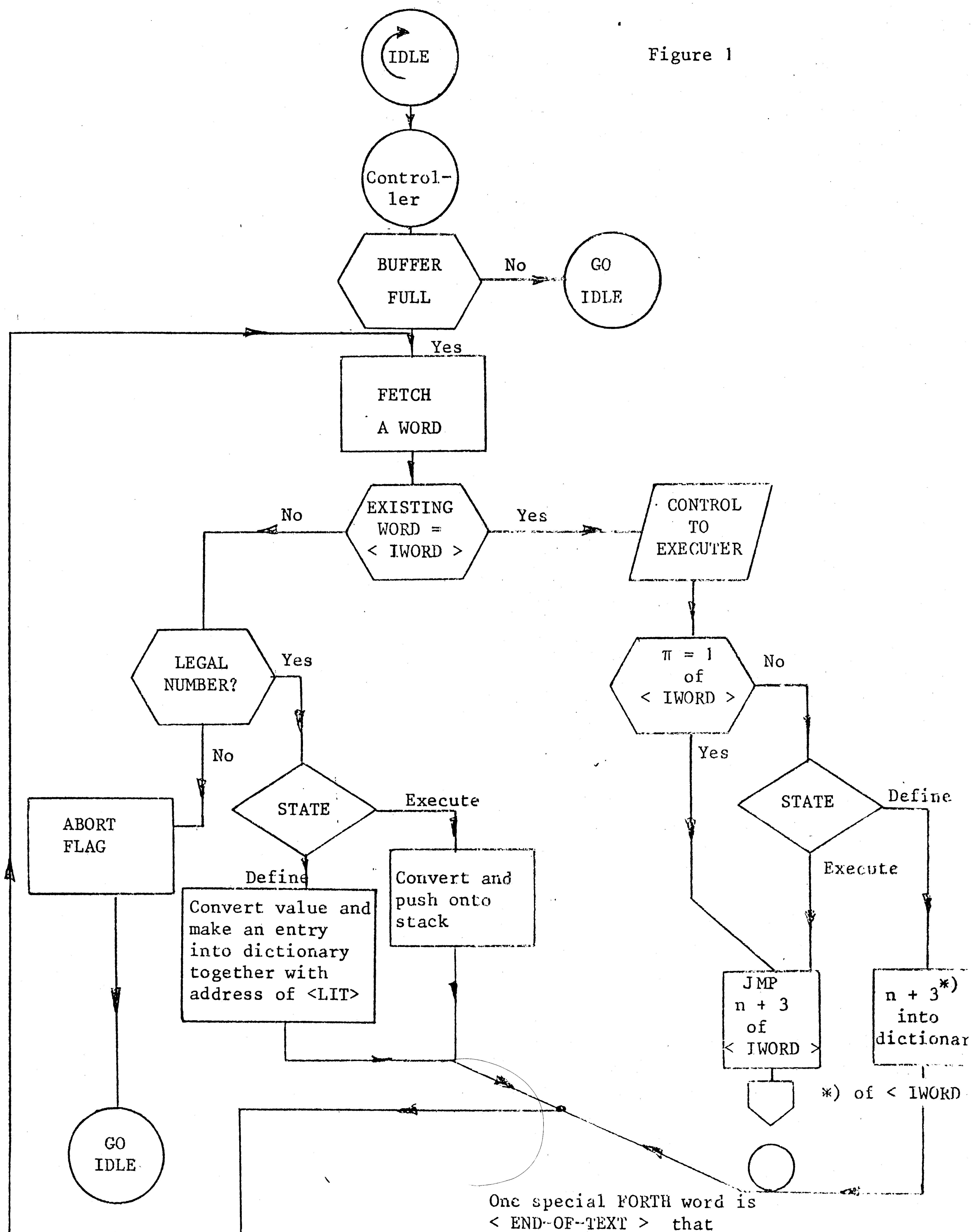
Ulam, reeks van 64-65
UPDATE 92

VECTOR 81
vergelijking 23-54

Vervolg

vocabulaires	25-48-51
VOCABULARY	48
Von Koch	38-106
WCH	90
wees	41
werkwoorden	87
WHERE	91
WHILE	52-53-62-64
XOR	23
[	32
]	32
↑	24

Schematic logical structure of FORTH computer operating control system



STERREWACHT
VDU-driver
Baudrate switch 6 kacor.
TRA